Capítulo 1

LA COMPLEJIDAD DE LOS ALGORITMOS

1.1 INTRODUCCIÓN

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. A tal método lo denominamos *algoritmo*.

En el presente texto nos vamos a centrar en dos aspectos muy importantes de los algoritmos, como son su diseño y el estudio de su eficiencia.

El primero se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema. Por otra parte, el segundo nos permite medir de alguna forma el coste (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema.

Este capítulo está dedicado al segundo de estos aspectos: la eficiencia. En cuanto a las técnicas de diseño, que corresponden a los patrones fundamentales sobre los que se construyen los algoritmos que resuelven un gran número de problemas, se estudiarán en los siguientes capítulos.

1.2 EFICIENCIA Y COMPLEJIDAD

Una vez dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

A menudo se piensa que un algoritmo sencillo no es muy eficiente. Sin embargo, la sencillez es una característica muy interesante a la hora de diseñar un algoritmo, pues facilita su verificación, el estudio de su eficiencia y su mantenimiento. De ahí que muchas veces prime la simplicidad y legibilidad del código frente a alternativas más crípticas y eficientes del algoritmo. Este hecho se pondrá de manifiesto en varios de los ejemplos mostrados a lo largo de este libro, en donde profundizaremos más en este compromiso.

Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el *espacio*, es decir, memoria que utiliza, y el *tiempo*, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de

entre varios que solucionan un mismo problema. En este capítulo nos centraremos solamente en la eficiencia temporal.

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida *teórica* (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
2. Y otro que ofrece una medida *real* (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son *funciones temporales* de los datos de entrada.

Entendemos por *tamaño de la entrada* el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar.

La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no puede ser expresada en segundos o en otra unidad de tiempo concreta, pues no existe un ordenador estándar al que puedan hacer referencia todas las medidas. Denotaremos por $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n.

Teóricamente $T(n)$ debe indicar el número de instrucciones ejecutadas por un ordenador idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar. Para ello es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2, que tardan $T_1(n)$ y $T_2(n)$ segundos respectivamente, el *Principio de Invarianza* afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo *del orden de* $T(n)$ si existen una constante real $c > 0$ y una implementación I del algoritmo que tarda menos que $cT(n)$, para todo n tamaño de la entrada.

Dos factores a tener muy en cuenta son la constante multiplicativa y el n_0 para los que se verifican las condiciones, pues si bien a priori un algoritmo de orden cuadrático es mejor que uno de orden cúbico, en el caso de tener dos algoritmos cuyos tiempos de ejecución son 10^6n^2 y $5n^3$ el primero sólo será mejor que el segundo para tamaños de la entrada superiores a 200.000.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenados que se encuentren ya los datos a ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta. Así suelen estudiarse tres casos para un mismo algoritmo: *caso peor*, *caso mejor* y *caso medio*.

El caso mejor corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones. Análogamente, el caso peor corresponde a la traza del algoritmo que realiza más instrucciones. Respecto al caso medio, corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

Es muy importante destacar que esos casos corresponden a un tamaño de la entrada dado, puesto que es un error común confundir el caso mejor con el que menos instrucciones realiza en cualquier caso, y por lo tanto contabilizar las instrucciones que hace para $n = 1$.

A la hora de medir el tiempo, siempre lo haremos en función del *número de operaciones elementales* que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa. También es posible distinguir entre los tiempos de ejecución de las diferentes operaciones elementales, lo cual es necesario a veces por las características específicas del ordenador (por ejemplo, se podría considerar que las operaciones + y ÷ presentan complejidades diferentes debido a su implementación). Sin embargo, en este texto tendremos en cuenta, a menos que se indique lo contrario, todas las operaciones elementales del lenguaje, y supondremos que sus tiempos de ejecución son todos iguales.

Para hacer un estudio del tiempo de ejecución de un algoritmo para los tres casos citados comenzaremos con un ejemplo concreto. Supongamos entonces que disponemos de la definición de los siguientes tipos y constantes:

```
CONST n =...; (* num. maximo de elementos de un vector *);
TYPE vector = ARRAY [1..n] OF INTEGER;
```

y de un algoritmo cuya implementación en Modula-2 es:

```
PROCEDURE Buscar(VAR a:vector;c:INTEGER):CARDINAL;
 VAR j:CARDINAL;
BEGIN
 j:=1;                          (* 1 *)
 WHILE (a[j]<c) AND (j<n) DO    (* 2 *)
    j:=j+1                      (* 3 *)
 END;                           (* 4 *)
 IF a[j]=c THEN                 (* 5 *)
    RETURN j                    (* 6 *)
 ELSE RETURN 0                  (* 7 *)
 END                            (* 8 *)
END Buscar;
```

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecuta 1 OE (una asignación).
- En la línea (2) se efectúa la condición del bucle, con un total de 4 OE (dos comparaciones, un acceso al vector, y un *AND*).
- La línea (3) está compuesta por un incremento y una asignación (2 OE).
- La línea (5) está formada por una condición y un acceso al vector (2 OE).
- La línea (6) contiene un *RETURN* (1 OE) si la condición se cumple.
- La línea (7) contiene un *RETURN* (1 OE), cuando la condición del *IF* anterior es falsa.

Obsérvese cómo no se contabiliza la copia del vector a la pila de ejecución del programa, pues se pasa por referencia y no por valor (está declarado como un argumento *VAR*, aunque no se modifique dentro de la función). En caso de pasarlo por valor, necesitaríamos tener en cuenta el coste que esto supone (un incremento de n OE). Con esto:

- En el *caso mejor* para el algoritmo, se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan de izquierda a derecha, y con "cortocircuito", es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia, $T(n)=1+2+3=6$.
- En el *caso peor*, se efectúa la línea (1), el bucle se repite $n-1$ veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. Por tanto

$$T(n) = 1 + \left(\left(\sum_{i=1}^{n-1} (4+2) \right) + 4 \right) + 2 + 1 = 6n + 2.$$

- En el *caso medio*, el bucle se ejecutará un número de veces entre 0 y n–1, y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder. Como existen n posibilidades (puede que el número buscado no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de $1/n$. Con esto, el número medio de veces que se efectuará el bucle es de

$$\sum_{i=0}^{n-1} i \frac{1}{n} = \frac{n-1}{2}.$$

  Tenemos pues que

$$T(n) = 1 + \left(\left(\sum_{i=1}^{(n-1)/2} (4+2) \right) + 2 \right) + 2 + 1 = 3n + 3.$$

Es importante observar que no es necesario conocer el propósito del algoritmo para analizar su tiempo de ejecución y determinar sus casos mejor, peor y medio, sino que basta con estudiar su código. Suele ser un error muy frecuente el determinar tales casos basándose sólo en la funcionalidad para la que el algoritmo fue concebido, olvidando que es el código implementado el que los determina.

En este caso, un examen más detallado de la función (¡y no de su nombre!) nos muestra que tras su ejecución, la función devuelve la posición de un entero dado c dentro de un vector ordenado de enteros, devolviendo 0 si el elemento no está en el vector. Lo que acabamos de probar es que su caso mejor se da cuando el elemento está en la primera posición del vector. El caso peor se produce cuando el elemento no está en el vector, y el caso medio ocurre cuando consideramos equiprobables cada una de las posiciones en las que puede encontrarse el elemento dentro del vector (incluyendo la posición especial 0, que indica que el elemento a buscar no se encuentra en el vector).

1.2.1 Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

- Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el Principio de Invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

- El tiempo de ejecución de la sentencia "`CASE C OF v1:S1|v2:S2|...|vn:Sn END;`" es $T = T(C) + \max\{T(S_1),T(S_2),...,T(S_n)\}$. Obsérvese que $T(C)$ incluye el tiempo de comparación con v_1, v_2 ,..., v_n.
- El tiempo de ejecución de la sentencia "`IF C THEN S1 ELSE S2 END;`" es $T = T(C) + \max\{T(S_1),T(S_2)\}$.
- El tiempo de ejecución de un bucle de sentencias "`WHILE C DO S END;`" es $T = T(C) +$ (nº iteraciones)$*(T(S) + T(C))$. Obsérvese que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.
- Para calcular el tiempo de ejecución del resto de sentencias iterativas (*FOR*, *REPEAT*, *LOOP*) basta expresarlas como un bucle *WHILE*. A modo de ejemplo, el tiempo de ejecución del bucle:

```
FOR i:=1 TO n DO
   S
END;
```

  puede ser calculado a partir del bucle equivalente:

```
i:=1;
WHILE i<=n DO
   S; INC(i)
END;
```

- El tiempo de ejecución de una llamada a un procedimiento o función $F(P_1, P_2,..., P_n)$ es 1 (por la llamada), más el tiempo de evaluación de los parámetros P_1, P_2,..., P_n, más el tiempo que tarde en ejecutarse F, esto es, $T = 1 + T(P_1) + T(P_2) + ... + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.
- También es necesario tener en cuenta, cuando el compilador las incorpore, las optimizaciones del código y la forma de evaluación de las expresiones, que pueden ocasionar "cortocircuitos" o realizarse de forma "perezosa" (*lazy*). En el presente trabajo supondremos que no se realizan optimizaciones, que existe el cortocircuito y que no existe evaluación perezosa.

1.3 COTAS DE COMPLEJIDAD. MEDIDAS ASINTÓTICAS

Una vez vista la forma de calcular el tiempo de ejecución T de un algoritmo, nuestro propósito es intentar clasificar dichas funciones de forma que podamos compararlas. Para ello, vamos a definir clases de equivalencia, correspondientes a las funciones que "crecen de la misma forma".

En las siguientes definiciones **N** denotará el conjunto de los números naturales y **R** el de los reales.

1.3.1 Cota Superior. Notación O

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan deprisa como f. Al conjunto de tales funciones se le llama cota superior de f y lo denominamos $O(f)$. Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

Definición 1.1

Sea f: $\mathbf{N}\rightarrow[0,\infty)$. Se define el conjunto de funciones de orden O (Omicron) de f como:

$$O(f) = \{g: \mathbf{N}\rightarrow[0,\infty) \mid \exists c\in \mathbf{R}, c>0, \exists n_0\in \mathbf{N} \bullet g(n) \leq cf(n) \ \ \forall n \geq n_0\}.$$

Diremos que una función t: $\mathbf{N} \rightarrow[0,\infty)$ es de orden O de f si $t \in O(f)$.

Intuitivamente, $t \in O(f)$ indica que t está acotada superiormente por algún múltiplo de f. Normalmente estaremos interesados en la menor función f tal que t pertenezca a $O(f)$.

En el ejemplo del algoritmo *Buscar* analizado anteriormente obtenemos que su tiempo de ejecución en el mejor caso es $O(1)$, mientras que sus tiempos de ejecución para los casos peor y medio son $O(n)$.

Propiedades de O

Veamos las propiedades de la cota superior. La demostración de todas ellas se obtiene aplicando la definición 1.1.

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \Rightarrow O(f) \subset O(g)$.
3. $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$.
4. Si $f \in O(g)$ y $g \in O(h) \Rightarrow f \in O(h)$.
5. Si $f \in O(g)$ y $f \in O(h) \Rightarrow f \in O(\min(g,h))$.
6. Regla de la suma: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g,h))$.
7. Regla del producto: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
8. Si existe $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$.
   b) Si $k = 0$ entonces $f \in O(g)$, es decir, $O(f) \subset O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

Obsérvese la importancia que tiene el que exista tal límite, pues si no existiese (o fuera infinito) no podría realizarse tal afirmación, como veremos en la resolución de los problemas de este capítulo.

De las propiedades anteriores se deduce que la relación $\sim_O$, definida por $f \sim_O g$ si y sólo si O(f) = O(g), es una relación de equivalencia. Siempre escogeremos el representante más sencillo para cada clase; así los órdenes de complejidad constante serán expresados por O(1), los lineales por O(n), etc.

1.3.2 Cota Inferior. Notación Ω

Dada una función f, queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. Al conjunto de tales funciones se le llama cota inferior de f y lo denominamos $\Omega(f)$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Definición 1.2

Sea f: $\mathbf{N}\rightarrow[0,\infty)$. Se define el conjunto de funciones de orden Ω (Omega) de f como:

$$\Omega(f) = \{g:\mathbf{N} \rightarrow [0,\infty) \mid \exists c\in \mathbf{R},\ c>0,\ \exists n_0\in \mathbf{N} \bullet g(n) \geq cf(n)\ \ \forall n \geq n_0\}.$$

Diremos que una función t:$\mathbf{N} \rightarrow[0,\infty)$ es de orden Ω de f si $t \in \Omega(f)$.

Intuitivamente, $t \in \Omega(f)$ indica que t está acotada inferiormente por algún múltiplo de f. Normalmente estaremos interesados en la mayor función f tal que t pertenezca a $\Omega(f)$, a la que denominaremos su cota inferior.

Obtener buenas cotas inferiores es en general muy difícil, aunque siempre existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior. Así, para ordenar n números una cota inferior sería n, y para multiplicar dos matrices de orden n sería n^2; sin embargo, los mejores algoritmos conocidos son de órdenes $n\log n$ y $n^{2.8}$ respectivamente.

Propiedades de Ω

Veamos las propiedades de la cota inferior Ω. La demostración de todas ellas se obtiene de forma simple aplicando la definición 1.2.

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \Rightarrow \Omega(f) \subset \Omega(g)$.
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$.
4. Si $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$.
5. Si $f \in \Omega(g)$ y $f \in \Omega(h) \Rightarrow f \in \Omega(\max(g,h))$.
6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$.

7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$.
8. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$.
   b) Si $k = 0$ entonces $g \in \Omega(f)$, es decir, $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $f \notin \Omega(g)$.

De las propiedades anteriores se deduce que la relación $\sim_\Omega$, definida por $f \sim_\Omega g$ si y sólo si $\Omega(f) = \Omega(g)$, es una relación de equivalencia. Al igual que hacíamos para el caso de la cota superior O, siempre escogeremos el representante más sencillo para cada clase. Así los órdenes de complejidad Ω constante serán expresados por $\Omega(1)$, los lineales por $\Omega(n)$, etc.

1.3.3 Orden Exacto. Notación Θ

Como última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

Definición 1.3

Sea f: $\mathbf{N}\to[0,\infty)$. Se define el conjunto de funciones de orden Θ (Theta) de f como:

$$\Theta(f) = O(f) \cap \Omega(f)$$

o, lo que es igual:

$$\Theta(f) = \{g:\mathbf{N}\to[0,\infty) \mid \exists c,d \in \mathbf{R},\ c,d>0,\ \exists n_0 \in \mathbf{N} \bullet cf(n) \le g(n) \le df(n)\ \forall n \ge n_0\}.$$

Diremos que una función t :$\mathbf{N}\to[0,\infty)$ es de orden Θ de f si $t \in \Theta(f)$.

Intuitivamente, $t \in \Theta(f)$ indica que t está acotada tanto superior como inferiormente por múltiplos de f, es decir, que t y f crecen de la misma forma.

Propiedades de Θ

Veamos las propiedades de la cota exacta. La demostración de todas ellas se obtiene también de forma simple aplicando la definición 1.3 y las propiedades de O y Ω.

1. Para cualquier función f se tiene que $f \in \Theta(f)$.
2. $f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$.
3. $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$.
4. Si $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$.
5. Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(\max(g,h))$.

6. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 \cdot f_2 \in \Theta(g \cdot h)$.
7. Si existe $\lim_{n\to\infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$.
   b) Si $k = 0$ los órdenes exactos de f y g son distintos.

1.3.4 Observaciones sobre las cotas asintóticas

1. La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño.
2. Para un algoritmo dado se pueden obtener tres funciones que miden su tiempo de ejecución, que corresponden a sus casos mejor, medio y peor, y que denominaremos respectivamente $T_m(n)$, $T_{1/2}(n)$ y $T_p(n)$. Para cada una de ellas podemos dar tres cotas asintóticas de crecimiento, por lo que se obtiene un total de nueve cotas para el algoritmo.
3. Para simplificar, dado un algoritmo diremos que su orden de complejidad es $O(f)$ si su tiempo de ejecución para el peor caso es de orden O de f, es decir, $T_p(n)$ es de orden $O(f)$. De forma análoga diremos que su orden de complejidad para el mejor caso es $\Omega(g)$ si su tiempo de ejecución para el mejor caso es de orden Ω de g, es decir, $T_m(n)$ es de orden $\Omega(g)$.
4. Por último, diremos que un algoritmo es de orden exacto $\Theta(f)$ si su tiempo de ejecución en el caso medio $T_{1/2}(n)$ es de este orden.

1.4 RESOLUCIÓN DE ECUACIONES EN RECURRENCIA

En las secciones anteriores hemos descrito cómo determinar el tiempo de ejecución de un algoritmo a partir del cómputo de sus operaciones elementales (OE). En general, este cómputo se reduce a un mero ejercicio de cálculo. Sin embargo, para los algoritmos recursivos nos vamos a encontrar con una dificultad añadida, pues la función que establece su tiempo de ejecución viene dada por una ecuación en recurrencia, es decir, $T(n) = E(n)$, en donde en la expresión E aparece la propia función T.

Resolver tal tipo de ecuaciones consiste en encontrar una expresión no recursiva de T, y por lo general no es una labor fácil. Lo que veremos en esta sección es cómo se pueden resolver algunos tipos concretos de ecuaciones en recurrencia, que son las que se dan con más frecuencia al estudiar el tiempo de ejecución de los algoritmos desarrollados según las técnicas aquí presentadas.

1.4.1 Recurrencias homogéneas

Son de la forma:

$$a_0 T(n) + a_1 T(n-1) + a_2 T(n-2) + ... + a_k T(n-k) = 0$$

donde los coeficientes a_i son números reales, y k es un número natural entre 1 y n. Para resolverlas vamos a buscar soluciones que sean combinaciones de funciones exponenciales de la forma:

$$T(n) = c_1 p_1(n) r_1^n + c_2 p_2(n) r_2^n + ... + c_k p_k(n) r_k^n = \sum_{i=1}^{k} c_i p_i(n) r_i^n ,$$

donde los valores $c_1, c_2, ..., c_n$ y $r_1, r_2, ..., r_n$ son números reales, y $p_1(n), ..., p_k(n)$ son polinomios en n con coeficientes reales. Si bien es cierto que estas ecuaciones podrían tener soluciones más complejas que éstas, se conjetura que serían del mismo orden y por tanto no nos ocuparemos de ellas.

Para resolverlas haremos el cambio $x^n = T(n)$, con lo cual obtenemos la *ecuación característica* asociada:

$$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + ... + a_k = 0 .$$

Llamemos $r_1, r_2, ..., r_k$ a sus raíces, ya sean reales o complejas. Dependiendo del orden de multiplicidad de tales raíces, pueden darse los dos siguientes casos.

Caso 1: Raíces distintas

Si todas las raíces de la ecuación característica son distintas, esto es, $r_i \neq r_j$ si $i \neq j$, entonces la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = c_1 r_1^n + c_2 r_2^n + ... + c_k r_k^n = \sum_{i=1}^{k} c_i r_i^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Como ejemplo, veamos lo que ocurre para la ecuación en recurrencia definida para la sucesión de Fibonacci:

$$T(n) = T(n-1) + T(n-2), \quad n \geq 2$$

con las condiciones iniciales $T(0) = 0$, $T(1) = 1$. Haciendo el cambio $x^2 = T(n)$ obtenemos su ecuación característica $x^2 = x + 1$, o lo que es igual, $x^2 - x - 1 = 0$, cuyas raíces son:

$$r_1 = \frac{1+\sqrt{5}}{2}, \quad r_2 = \frac{1-\sqrt{5}}{2}$$

y por tanto

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^n .$$

Para calcular las constantes c_1 y c_2 necesitamos utilizar las condiciones iniciales de la ecuación original, obteniendo:

$$\left.\begin{array}{l} T(0) = c_1\left(\dfrac{1+\sqrt{5}}{2}\right)^0 + c_2\left(\dfrac{1-\sqrt{5}}{2}\right)^0 = c_1 + c_2 = 0 \\ T(1) = c_1\left(\dfrac{1+\sqrt{5}}{2}\right)^1 + c_2\left(\dfrac{1-\sqrt{5}}{2}\right)^1 = 1 \end{array}\right\} \Rightarrow c_1 = -c_2 = \frac{1}{\sqrt{5}}.$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \in \mathrm{O}(\varphi^n).$$

Caso 2: Raíces con multiplicidad mayor que 1

Supongamos que alguna de las raíces (p.e. r_1) tiene multiplicidad m>1. Entonces la ecuación característica puede ser escrita en la forma

$$(x - r_1)^m(x - r_2)...(x - r_{k-m+1})$$

en cuyo caso la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m} c_i n^{i-1} r_1^n + \sum_{i=m+1}^{k} c_i r_{i-m+1}^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Veamos un ejemplo en el que la ecuación en recurrencia es:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), \quad n \geq 2$$

con las condiciones iniciales $T(k) = k$ para $k = 0, 1, 2$. La ecuación característica que se obtiene es $x^3 - 5x^2 + 8x - 4 = 0$, o lo que es igual $(x-2)^2(x-1) = 0$ y por tanto,

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 1^n.$$

De las condiciones iniciales obtenemos $c_1 = 2$, $c_2 = -1/2$ y $c_3 = -2$, por lo que

$$T(n) = 2^{n+1} - n2^{n-1} - 2 \in \Theta(n2^n).$$

Este caso puede ser generalizado de la siguiente forma. Si $r_1, r_2, ..., r_k$ son las raíces de la ecuación característica de una ecuación en recurrencia homogénea, cada una de multiplicidad m_i, esto es, si la ecuación característica puede expresarse como:

$$(x - r_1)^{m_1}(x - r_2)^{m_2}...(x - r_k)^{m_k} = 0,$$

entonces la solución a la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m_1} c_{1i} n^{i-1} r_1^n + \sum_{i=1}^{m_2} c_{2i} n^{i-1} r_2^n + \ldots + \sum_{i=1}^{m_k} c_{ki} n^{i-1} r_k^n \,.$$

1.4.2 Recurrencias no homogéneas

Consideremos una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = b^n p(n)$$

donde los coeficientes a_i y b son números reales, y $p(n)$ es un polinomio en n de grado d. Una primera idea para resolver la ecuación es manipularla para convertirla en homogénea, como muestra el siguiente ejemplo.

Sea la ecuación $T(n) - 2T(n-1) = 3^n$ para $n \geq 2$, con las condiciones iniciales $T(0) = 0$ y $T(1) = 1$. En este caso $b = 3$ y $p(n) = 1$, polinomio en n de grado 0.

Podemos escribir la ecuación de dos formas distintas. En primer lugar, para n+1 tenemos que

$$T(n+1) - 2T(n) = 3^{n+1}.$$

Pero si multiplicamos por 3 la ecuación original obtenemos:

$$3T(n) - 6T(n-1) = 3^{n+1}$$

Restando ambas ecuaciones, conseguimos

$$T(n+1) - 5T(n) + 6T(n-1) = 0,$$

que resulta ser una ecuación homogénea cuya solución, aplicando lo visto anteriormente, es

$$T(n) = 3^n - 2^n \in \Theta(3^n).$$

Estos cambios son, en general, difíciles de ver. Afortunadamente, para este tipo de ecuaciones también existe una fórmula general para resolverlas, buscando sus soluciones entre las funciones que son combinaciones lineales de exponenciales, en donde se demuestra que la ecuación característica es de la forma:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \ldots + a_k)(x-b)^{d+1} = 0\,,$$

lo que permite resolver el problema de forma similar a los casos anteriores.

Como ejemplo, veamos cómo se resuelve la ecuación en recurrencia que plantea el algoritmo de las torres de Hanoi:

$$T(n) = 2T(n-1) + n.$$

Su ecuación característica es entonces $(x-2)(x-1)^2 = 0$, y por tanto

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n \in \Theta(2^n).$$

Generalizando este proceso, supongamos ahora una ecuación de la forma:

$$a_0T(n)+a_1T(n-1)+...+a_kT(n-k)=b_1^n p_1(n)+b_2^n p_2(n)+...+b_s^n p_s(n)$$

donde como en el caso anterior, los coeficientes a_i y b_i son números reales y $p_j(n)$ son polinomios en n de grado d_j. En este caso también existe una forma general de la solución, en donde se demuestra que la ecuación característica es:

$$(a_0x^k + a_1x^{k-1} + a_2x^{k-2} + ... + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1}...(x-b_s)^{d_s+1} = 0\,.$$

Como ejemplo, supongamos la ecuación

$$T(n) = 2T(n–1) + n + 2^n,\ n\geq1,$$

con la condición inicial $T(0) = 1$. En este caso tenemos que $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ y $p_2(n) = 1$, por lo que su ecuación característica es $(x–2)^2(x–1)^2 = 0$, lo que da lugar a la expresión final de $T(n)$:

$$T(n) = –2 – n + 2^{n+1} + n2^n \in \Theta(n2^n).$$

1.4.3 Cambio de Variable

Esta técnica se aplica cuando n es potencia de un número real a, esto es, $n = a^k$. Sea por ejemplo, para el caso $a = 2$, la ecuación $T(n) = 4T(n/2) + n$, donde n es una potencia de 2 ($n > 3$), $T(1) = 1$, y $T(2) = 6$.
Si $n = 2^k$ podemos escribir la ecuación como:

$$T(2^k) = 4T(2^{k–1}) + 2^k.$$

Haciendo el cambio de variable $t_k = T(2^k)$ obtenemos la ecuación

$$t_k = 4t_{k–1} + 2^k$$

que corresponde a una de las ecuaciones estudiadas anteriormente, cuya solución viene dada por la expresión

$$t_k = c_1(2^k)^2 + c_22^k.$$

Deshaciendo el cambio que realizamos al principio obtenemos que

$$T(n) = c_1n^2 + c_2n.$$

Calculando entonces las constantes a partir de las condiciones iniciales:

$$T(n) = 2n^2 – n \in \Theta(n^2).$$

1.4.4 Recurrencias No Lineales

En este caso, la ecuación que relaciona $T(n)$ con el resto de los términos no es lineal. Para resolverla intentaremos convertirla en una ecuación lineal como las que hemos estudiado hasta el momento.

Por ejemplo, sea la ecuación $T(n) = nT^2(n/2)$ para n potencia de 2, $n > 1$, con la condición inicial $T(1) = 1/3$. Llamando $t_k = T(2^k)$, la ecuación queda como

$$t_k = T(2^k) = 2^k T^2(2^{k-1}) = 2^k t_{k-1}^2,$$

que no corresponde a ninguno de los tipos estudiados. Necesitamos hacer un cambio más para transformar la ecuación. Tomando logaritmos a ambos lados y haciendo el cambio $u_k = \log t_k$ obtenemos

$$u_k - 2u_{k-1} = k,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$u_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $u_k = \log t_k$

$$t_k = 2^{c_1 2^k + c_2 + c_3 k}$$

y después $t_k = T(2^k)$. En consecuencia

$$T(n) = 2^{c_1 n + c_2 + c_3 \log n}.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia original para obtener las restantes:

$$T(2) = 2T^2(1) = 2/9.$$

$$T(4) = 4T^2(2) = 16/81.$$

Con esto llegamos a que $c_1 = \log(4/3) = 2 - \log 3$, $c_2 = -2$, $c_3 = -1$ y por consiguiente:

$$T(n) = \frac{2^{2n}}{4n3^n}.$$

1.5 PROBLEMAS PROPUESTOS

1.1. De las siguientes afirmaciones, indicar cuales son ciertas y cuales no:

(*i*) $n^2 \in O(n^3)$
(*ii*) $n^3 \in O(n^2)$
(*iii*) $2^{n+1} \in O(2^n)$
(*iv*) $(n+1)! \in O(n!)$
(*v*) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$
(*vi*) $3^n \in O(2^n)$
(*vii*) $\log n \in O(n^{1/2})$
(*viii*) $n^{1/2} \in O(\log n)$
(*ix*) $n^2 \in \Omega(n^3)$
(*x*) $n^3 \in \Omega(n^2)$
(*xi*) $2^{n+1} \in \Omega(2^n)$
(*xii*) $(n+1)! \in \Omega(n!)$
(*xiii*) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$
(*xiv*) $3^n \in \Omega(2^n)$
(*xv*) $\log n \in \Omega(n^{1/2})$
(*xvi*) $n^{1/2} \in \Omega(\log n)$

1.2. Sea a una constante real, $0<a<1$. Usar las relaciones $\subset$ y $=$ para ordenar los órdenes de complejidad de las siguientes funciones: $n\log n$, $n^2\log n$, n^8, n^{1+a}, $(1+a)^n$, $(n^2+8n+\log^3 n)^4$, $n^2/\log n$, 2^n.

1.3. La siguiente ecuación recurrente representa un caso típico de un algoritmo recursivo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n \le b \\ aT(n-b) + cn^k & \text{si } n > b \end{cases}$$

donde a,c,k son números reales, n,b son números naturales, y $a>0$, $c>0$, $k\geq 0$. En general, la constante a representa el número de llamadas recursivas que se realizan para un problema de tamaño n en cada ejecución del algoritmo; $n–b$ es el tamaño de los subproblemas generados; y cn^k representa el coste de las instrucciones del algoritmo que no son llamadas recursivas.

Demostrar que $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \operatorname{div} b}) & \text{si } a > 1 \end{cases}$

1.4. La siguiente ecuación recurrente representa un caso típico de *Divide y Vencerás*:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \le n < b \\ aT(n/b) + cn^k & \text{si } n \ge b \end{cases}$$

donde a,c,k son números reales, n,b son números naturales, y $a>0$, $c>0$, $k\geq 0$, $b>1$. La expresión cn^k representa en general el coste de descomponer el problema inicial en a subproblemas y el de componer las soluciones para producir la solución del problema original.

Demostrar que $T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$

1.5. Supongamos que disponemos de la siguiente definición de tipo:

```
CONST n = ...;
TYPE vector = ARRAY [1..n] OF INTEGER;
```

Consideramos entonces los procedimientos y funciones siguientes:

```
PROCEDURE Algoritmo1(VAR a:vector);
 VAR i,j:CARDINAL;
     temp:INTEGER;
BEGIN
 FOR i:=1 TO n-1 DO                     (* 1 *)
   FOR j:=n TO i+1 BY -1 DO             (* 2 *)
     IF a[j-1]>a[j] THEN                (* 3 *)
       temp:=a[j-1];                    (* 4 *)
       a[j-1]:=a[j];                    (* 5 *)
       a[j]:=temp                       (* 6 *)
     END                                (* 7 *)
   END                                  (* 8 *)
 END                                    (* 9 *)
END Algoritmo1;


PROCEDURE Algoritmo2(VAR a:vector;c:INTEGER):CARDINAL;
 VAR inf,sup,i:CARDINAL;
BEGIN
 inf:=1; sup:=n;                        (* 1 *)
 WHILE (sup>=inf) DO                    (* 2 *)
   i:=(inf+sup) DIV 2;                  (* 3 *)
   IF a[i]=c THEN RETURN i              (* 4 *)
   ELSIF c<a[i] THEN sup:=i-1           (* 5 *)
   ELSE inf:=i+1                        (* 6 *)
   END                                  (* 7 *)
 END;                                   (* 8 *)
 RETURN 0;                              (* 9 *)
END Algoritmo2;
```

```
PROCEDURE Euclides(m,n:CARDINAL):CARDINAL;
 VAR temp:CARDINAL;
BEGIN
 WHILE m>0 DO                            (* 1 *)
   temp:=m;                              (* 2 *)
   m:=n MOD m;                           (* 3 *)
   n:=temp                               (* 4 *)
 END;                                    (* 5 *)
 RETURN n                                (* 6 *)
END Euclides;
```

```
PROCEDURE Misterio(n:CARDINAL);
 VAR i,j,k,s:INTEGER;
BEGIN
 s:=0;                                   (* 1 *)
 FOR i:=1 TO n-1 DO                      (* 2 *)
   FOR j:=i+1 TO n DO                    (* 3 *)
     FOR k:=1 TO j DO                    (* 4 *)
       s:=s+2                            (* 5 *)
     END                                 (* 6 *)
   END                                   (* 7 *)
 END                                     (* 8 *)
END Misterio;
```

a) Calcular sus tiempos de ejecución en el mejor, peor, y caso medio.
b) Dar cotas asintóticas O, Ω y Θ para las funciones anteriores.

1.6. Demostrar las siguientes inclusiones estrictas: $O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!)$.

1.7. a) Demostrar que $f \in O(g) \Leftrightarrow g \in \Omega(f)$.
b) Dar un ejemplo de funciones f y g tales que $f \in O(g)$ pero que $f \notin \Omega(g)$.
c) Demostrar que $\forall a,b>1$ se tiene que $\log_a n \in \Theta(\log_b n)$.

1.8. Considérense las siguientes funciones de n:

$f_1(n) = n^2$; $f_2(n) = n^2 + 1000n$;

$$f_3(n) = \begin{cases} n, & \text{si } n \text{ impar} \\ n^3, & \text{si } n \text{ par} \end{cases}; \quad f_4(n) = \begin{cases} n, & \text{si } n \le 100 \\ n^3, & \text{si } n > 100 \end{cases}$$

Para cada posible valor de i,j indicar si $f_i \in O(f_j)$ y si $f_i \in \Omega(f_j)$.

1.9. Resolver las siguientes ecuaciones y dar su orden de complejidad:

a) $T(n)=3T(n-1)+4T(n-2)$ si $n>1$; $T(0)=0$; $T(1)=1$.

b) $T(n)=2T(n-1)-(n+5)3^n$ si $n>0$; $T(0)=0$.

c) $T(n)=4T(n/2)+n^2$ si $n>4$, n potencia de 2; $T(1)=1$; $T(2)=8$.

d) $T(n)=2T(n/2)+n\log n$ si $n>1$, n potencia de 2.

e) $T(n)=3T(n/2)+5n+3$ si $n>1$, n potencia de 2.

f) $T(n)=2T(n/2)+\log n$ si $n>1$, n potencia de 2.

g) $T(n)=2T(n^{1/2})+\log n$ con $n=2^{2^k}$; $T(2)=1$.

h) $T(n)=5T(n/2)+(n\log n)^2$ si $n>1$, n potencia de 2; $T(1)=1$.

i) $T(n)=T(n-1)+2T(n-2)-2T(n-3)$ si $n>2$; $T(n)=9n^2-15n+106$ si $n=0,1,2$.

j) $T(n)=(3/2)T(n/2)-(1/2)T(n/4)-(1/n)$ si $n>2$; $T(1)=1$; $T(2)=3/2$.

k) $T(n)=2T(n/4)+n^{1/2}$ si $n>4$, n potencia de 4.

l) $T(n)=4T(n/3)+n^2$ si $n>3$, n potencia de 3.

1.10. Suponiendo que $T_1 \in O(f)$ y que $T_2 \in O(f)$, indicar cuáles de las siguientes afirmaciones son ciertas:

a) $T_1 + T_2 \in O(f)$.

b) $T_1 - T_2 \in O(f)$.

c) $T_1 / T_2 \in O(1)$.

d) $T_1 \in O(T_2)$.

1.11. Encontrar dos funciones $f(n)$ y $g(n)$ tales que $f \notin O(g)$ y $g \notin O(f)$.

1.12. Demostrar que para cualquier constante k se verifica que $\log^k n \in O(n)$.

1.13. Consideremos los siguientes procedimientos y funciones sobre árboles. Calcular sus tiempos de ejecución y sus órdenes de complejidad.

```
PROCEDURE Inorden(t:arbol);    (* recorrido en inorden de t *)
BEGIN
 IF NOT Esvacio(t) THEN                                  (* 1
*)
    Inorden(Izq(t));                                    (* 2 *)
    Opera(Raiz(t));                                     (* 3 *)
    Inorden(Der(t))                                     (* 4 *)
 END;                                                   (* 5 *)
END Inorden;
```

```
PROCEDURE Altura(t:arbol):CARDINAL; (* altura de t *)
BEGIN
 IF Esvacio(t) THEN                                    (* 1 *)
   RETURN 0                                            (* 2 *)
 ELSE                                                  (* 3 *)
   RETURN 1+Max2(Altura(Izq(t)),Altura(Der(t)))        (* 4 *)
 END                                                   (* 5 *)
END Altura;

PROCEDURE Mezcla(t1,t2:arbol):arbol;
(* devuelve un arbol binario de busqueda con los elementos de
   los dos arboles binarios de busqueda t1 y t2. La funcion Ins
   inserta un elemento en un arbol binario de busqueda *)
BEGIN
 IF Esvacio(t1) THEN                                   (* 1 *)
  RETURN t2                                            (* 2 *)
 ELSIF Esvacio(t2) THEN                                (* 3 *)
  RETURN t1                                            (* 4 *)
 ELSE                                                  (* 5 *)
   RETURN Mezcla(Mezcla(Ins(t1,Raiz(t2)),Izq(t2)),
                Der(t2))                               (* 6 *)
 END                                                   (* 7 *)
END Mezcla;
```

Supondremos que las operaciones básicas del tipo abstracto de datos *arbol* (*Raiz*, *Izq*, *Der*, *Esvacio*) son O(1), así como las operaciones *Opera* (que no es relevante lo que hace) y *Max2* (que calcula el máximo de dos números). Por otro lado, supondremos que la complejidad de la función *Ins* es O(log*n*).

1.14. Ordenar las siguientes funciones de acuerdo a su velocidad de crecimiento: n, $\sqrt{n}$, $\log n$, $\log\log n$, $\log^2 n$, $n/\log n$, $\sqrt{n}\log^2 n$, $(1/3)^n$, $(3/2)^n$, 17, n^2.

1.15. Resolver la ecuación $T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i)\right) + cn$, siendo $T(0) = 0$.

1.16. Consideremos las siguientes funciones:

```
CONST n = ...;
TYPE vector = ARRAY[1..n] OF INTEGER;
```

```
PROCEDURE BuscBin(VAR a:vector;
                  prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
 VAR mitad:CARDINAL;
BEGIN
 IF (prim>=ult) THEN RETURN a[ult]=x              (* 1 *)
 ELSE                                             (* 2 *)
    mitad:=(prim+ult)DIV 2;                       (* 3 *)
    IF x=a[mitad] THEN RETURN TRUE                 (* 4 *)
    ELSIF (x<a[mitad]) THEN                       (* 5 *)
       RETURN BuscBin(a,prim,mitad-1,x)           (* 6 *)
    ELSE                                          (* 7 *)
       RETURN BuscBin(a,mitad+1,ult,x)            (* 8 *)
    END                                           (* 9 *)
 END                                              (* 10 *)
END BuscBin;

PROCEDURE Sumadigitos(num:CARDINAL):CARDINAL;
BEGIN
 IF num<10 THEN RETURN num                        (* 1 *)
 ELSE RETURN (num MOD 10)+Sumadigitos(num DIV 10) (* 2 *)
 END                                              (* 3 *)
END Sumadigitos;
```

a) Calcular sus tiempos de ejecución y sus órdenes de complejidad.
b) Modificar los algoritmos eliminando la recursión.
c) Calcular la complejidad de los algoritmos modificados y justificar para qué casos es más conveniente usar uno u otro.

1.17. Consideremos la siguiente función:

```
PROCEDURE Raro(VAR a:vector;prim,ult:CARDINAL):INTEGER;
 VAR mitad,terc:CARDINAL;
BEGIN
 IF (prim>=ult) THEN RETURN a[ult] END;
 mitad:=(prim+ult)DIV 2;    (* posicion central *)
 terc :=(ult-prim)DIV 3;    (* num. elementos DIV 3 *)
 RETURN a[mitad]+Raro(a,prim,prim+terc)+Raro(a,ult-terc,ult)
END Raro;
```

a) Calcular el tiempo de ejecución de la llamada a la función *Raro(a,*1*,n)*, suponiendo que *n* es potencia de 3.
b) Dar una cota de complejidad para dicho tiempo de ejecución.

1.6 SOLUCIÓN A LOS PROBLEMAS PROPUESTOS

Antes de comenzar con la resolución de los problemas es necesario hacer una aclaración sobre la notación utilizada para las funciones logarítmicas. A partir de ahora y a menos que se exprese explícitamente otra base, la función "log" hará referencia a logaritmos en base dos.

Solución al Problema 1.1 (☺/☹)

(i) $n^2 \in O(n^3)$ es cierto pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(ii) $n^3 \in O(n^2)$ es falso pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(iii) $2^{n+1} \in O(2^n)$ es cierto pues $lim_{n\to\infty} (2^{n+1}/2^n) = 2$.

(iv) $(n+1)! \in O(n!)$ es falso pues $lim_{n\to\infty} (n!/(n+1)!) = 0$.

(v) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$ es falso. Por ejemplo, sea $f(n) = 3n$; claramente $f(n) \in O(n)$ pero sin embargo $lim_{n\to\infty} (2^n/2^{3n}) = 0$, con lo cual $2^{3n} \notin O(2^n)$. De forma más general, resulta ser falso para cualquier función lineal de la forma $f(n) = \alpha n$ con $\alpha > 1$, y cierto para $f(n) = \beta n$ con $\beta \leq 1$.

(vi) $3^n \in O(2^n)$ es falso pues $lim_{n\to\infty} (2^n/3^n) = 0$.

(vii) $\log n \in O(n^{1/2})$ es cierto pues $lim_{n\to\infty} (\log n/n^{1/2}) = 0$.

(viii) $n^{1/2} \in O(\log n)$ es falso pues $lim_{n\to\infty} (\log n/n^{1/2}) = 0$.

(ix) $n^2 \in \Omega(n^3)$ es falso pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(x) $n^3 \in \Omega(n^2)$ es cierto pues $lim_{n\to\infty} (n^2/n^3) = 0$.

(xi) $2^{n+1} \in \Omega(2^n)$ es cierto pues $lim_{n\to\infty} (2^{n+1}/2^n) = 2$.

(xii) $(n+1)! \in \Omega(n!)$ es cierto pues $lim_{n\to\infty} (n!/(n+1)!) = 0$.

(xiii) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$ es falso. Por ejemplo, sea $f(n) = (1/2)n$; claramente $f(n) \in O(n)$ pero sin embargo $lim_{n\to\infty} (2^{(1/2)n}/2^n) = 0$, con lo cual $2^{(1/2)n} \notin \Omega(2^n)$. De forma más general, resulta ser falso para cualquier función $f(n) = \alpha n$ con $\alpha < 1$, y cierto para $f(n) = \beta n$ con $\beta \geq 1$.

(xiv) $3^n \in \Omega(2^n)$ es cierto pues $lim_{n\to\infty} (2^n/3^n) = 0$.

(xv) $\log n \in \Omega(n^{1/2})$ es falso pues $lim_{n\to\infty}(\log n/n^{1/2}) = 0$.

(xvi) $n^{1/2} \in \Omega(\log n)$ es cierto pues $lim_{n\to\infty}(\log n/n^{1/2}) = 0$.

Solución al Problema 1.2 (☺)

- Respecto al orden de complejidad O tenemos que:

$$O(n\log n) \subset O(n^{1+a}) \subset O(n^2/\log n) \subset O(n^2\log n) \subset O(n^8) = O((n^2+8n+\log^3 n)^4) \subset O((1+a)^n) \subset O(2^n).$$

  Puesto que todas las funciones son continuas, para comprobar que $O(f)\subset O(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n)) = 0$, y para comprobar que $O(f) = O(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n))$ es finito y distinto de 0.

- Por otro lado, respecto al orden de complejidad Ω, obtenemos que:

$$\Omega(n\log n) \supset \Omega(n^{1+a}) \supset \Omega(n^2/\log n) \supset \Omega(n^2\log n) \supset \Omega(n^8) = \Omega((n^2+8n+\log^3 n)^4) \supset \Omega((1+a)^n) \supset \Omega(2^n)$$

  Para comprobar que $\Omega(f) \subset \Omega(g)$, basta ver que $lim_{n\to\infty}(g(n)/f(n)) = 0$, y para comprobar que $\Omega(f) = \Omega(g)$, basta ver que $lim_{n\to\infty}(f(n)/g(n))$ es finito y distinto de 0 puesto que al ser las funciones continuas tenemos garantizada la existencia de los límites.

- Y en lo relativo al orden de complejidad Θ, al definirse como la intersección de los órdenes O y Ω, sólo tenemos asegurado que:

$$\Theta(n^8) = \Theta((n^2+8n+\log^3 n)^4),$$

  siendo los órdenes Θ del resto de las funciones conjuntos no comparables.

Solución al Problema 1.3 (✍)

La ecuación dada puede ser también escrita como $T(n) - aT(n-b) = cn^k$, ecuación en recurrencia no homogénea cuya ecuación característica es:

$$(x^b-a)(x-1)^{k+1} = 0.$$

- Para estudiar las raíces de esa ecuación, vamos a suponer primero que $a \neq 1$. En este caso, la ecuación tiene una raíz de multiplicidad k+1 (el 1), y b raíces

distintas $r_1, r_2, ..., r_b$ (las b raíces b-ésimas de a†). Entonces la solución de la ecuación en recurrencia es de la forma:

$$T(n) = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + ... + c_{k+1} n^k 1^n + d_1 r_1^n + d_2 r_2^n + ... + d_b r_b^n =$$
$$= \left(\sum_{i=1}^{k+1} c_i n^{i-1} \right) + \left(\sum_{i=1}^{b} d_i r_i^n \right)$$

siendo c_i y d_i coeficientes reales.

– Si $a<1$, las raíces b-ésimas de a (esto es, las r_i) son menores en módulo que 1, con lo cual el segundo sumatorio tiende a cero cuando n tiende a ∞, y en consecuencia $T(n) \in \Theta(n^k)$ pues $\lim_{n\to\infty} \frac{T(n)}{n^k} = c_{k+1}$ es finito y distinto de cero.

  Para ver que efectivamente c_{k+1} es distinto de cero independientemente de las condiciones iniciales, sustituimos esta expresión de $T(n)$ en la ecuación original, $T(n) - aT(n-b) = cn^k$, y por tanto:

$$\left(\sum_{i=1}^{k+1} c_i n^{i-1} + \sum_{i=1}^{b} d_i r_i^n \right) - a\left(\sum_{i=1}^{k+1} c_i (n-b)^{i-1} + \sum_{i=1}^{b} d_i r_i^n \right) = cn^k .$$

  Igualando ahora los coeficientes que acompañan a n^k obtenemos que $c_{k+1} - ac_{k+1} = c$, o lo que es igual, $(1-a)c_{k+1} = c$. Ahora bien, como sabemos que $a < 1$ y $c > 0$, entonces c_{k+1} no puede ser cero.

– Si $a > 1$, las funciones del segundo sumatorio son exponenciales, mientras que las primeras se mantienen dentro de un orden polinomial, por lo que en este caso el orden de complejidad del algoritmo es exponencial. Ahora bien, como todas las raíces b-ésimas de a tienen el mismo módulo, todas crecen de la misma forma y por tanto todas son del mismo orden de complejidad, obteniendo que

$$\Theta(r_1^n) = \Theta(r_2^n) = ... = \Theta(r_b^n) .$$

  Como $\lim_{n\to\infty} \frac{T(n)}{r_1^n} = d_1$ es distinto de cero y finito, podemos concluir que:

$$T(n) \in \Theta(r_1^n) = \Theta\left(\left(\sqrt[b]{a}\right)^n \right) = \Theta(a^{n \operatorname{div} b}) .$$

Hemos supuesto que $d_1 \neq 0$. Esto no tiene por qué ser necesariamente cierto para todas las condiciones iniciales, aunque sin embargo sí es cierto que al menos uno de los coeficientes d_i ha de ser distinto de cero. Basta tomar ese sumando para demostrar lo anterior.

---

† Recordemos que dados dos números reales a y b, la solución de la ecuación $x^b - a = 0$ tiene b raíces distintas, que pueden ser expresadas como $a^{1/b}e^{2\pi ik/n}$, para $k=0,1,2,...,n-1$.

- Supongamos ahora que $a = 1$. En este caso la multiplicidad de la raíz 1 es $k+2$, con lo cual

$$T(n) = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + ... + c_{k+2} n^{k+1} 1^n + d_2 r_2^n + ... + d_b r_b^n =$$
$$= \left(\sum_{i=1}^{k+2} c_i n^{i-1} \right) + \left(\sum_{i=2}^{b} d_i r_i^n \right)$$

Pero las raíces $r_2, r_3, ..., r_b$ son todas de módulo 1 (obsérvese que $r_1=1$), y por tanto el segundo sumando de $T(n)$ es de complejidad $\Theta(1)$.

Así, el crecimiento de $T(n)$ coincide con el del primer sumando, que es un polinomio de grado $k+1$ con lo cual $T(n) \in \Theta(n^{k+1})$.

Solución al Problema 1.4 (☞)

Haciendo el cambio $n = b^m$, o lo que es igual, $m = \log_b n$, obtenemos que

$$T(b^m) = aT(b^{m-1}) + cb^{mk}.$$

Llamando $t_m = T(b^m)$, la ecuación queda como

$$t_m - at_{k-1} = c(b^k)^m,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-a)(x-b^k) = 0$.

Para resolver esta ecuación, supongamos primero que $a = b^k$. Entonces, la ecuación característica es $(x-b^k)^2 = 0$ y por tanto

$$t_m = c_1 b^{km} + c_2 m b^{km}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_m = T(b^m)$ con lo que

$$T(b^m) = c_1 b^{km} + c_2 m b^{km} = (c_1 + c_2 m) b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = (c_1 + c_2 \log_b n) n^k \in \Theta(n^k \log n).^‡$$

Supongamos ahora el caso contrario, $a \neq b^k$. Entonces la ecuación característica tiene dos raíces distintas, y por tanto

$$t_m = c_1 a^m + c_2 b^{km}.$$

Necesitamos deshacer los cambios hechos. Primero $t_m = T(b^m)$, con lo que

$$T(b^m) = c_1 a^m + c_2 b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = c_1 a^{\log_b n} + c_2 n^k = c_1 n^{\log_b a} + c_2 n^k.$$

‡ Obsérvese que se hace uso de que $\log_b n \in \Theta(\log n)$, lo que se demuestra en el problema 1.7.

En consecuencia, si $\log_b a > k$ (si y sólo si $a > b^k$) entonces $T(n) \in \Theta(n^{\log_b a})$. Si no, es decir, $a < b^k$, entonces $T(n) \in \Theta(n^k)$.

Solución al Problema 1.5

Procedimiento *Algoritmo1* (☺)

a) Para obtener el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
– Igual ocurre con la línea (2), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
– En la línea (3) se efectúa una condición, con un total de 4 OE (una diferencia, dos accesos a un vector, y una comparación).
– Las líneas (4) a (6) sólo se ejecutan si se cumple la condición de la línea (3), y realizan un total de 9 OE: 3, 4 y 2 respectivamente.

Con esto:

En el *caso mejor* para el algoritmo la condición de la línea (3) será siempre falsa, y no se ejecutarán nunca las líneas (4), (5) y (6). Así, el bucle más interno realizará $(n–i)$ iteraciones, cada una de ellas con 4 OE (línea 3), más las 3 OE de la línea (2). Por tanto, el bucle más interno realiza un total de

$$\left(\sum_{j=i+1}^{n}(4+3)\right)+3=7\left(\sum_{j=i+1}^{n}1\right)+3=7(n-i)+3$$

OE, siendo el 3 adicional por la condición de salida del bucle.

A su vez, el bucle externo repetirá esas $7(n–i)+3$ OE en cada iteración, lo que hace que el número de OE que se realizan en el algoritmo sea:

$$T(n)=\left(\sum_{i=1}^{n-1}\big(7(n-i)+3\big)+3\right)+3=\frac{7}{2}n^2+\frac{5}{2}n-3.$$

- En el *caso peor*, la condición de la línea (3) será siempre verdadera, y las líneas (4), (5) y (6) se ejecutarán en todas las iteraciones. Por tanto, el bucle más interno realiza

$$\left(\sum_{j=i+1}^{n}(4+9+3)\right)+3=16(n-i)+3$$

OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

$$T(n) = \left(\sum_{i=1}^{n-1} \big(16(n-i)+3\big)+3 \right) + 3 = 8n^2 - 2n - 3 .$$

- En el *caso medio*, la condición de la línea (3) será verdadera con probabilidad 1/2. Así, las líneas (4), (5) y (6) se ejecutarán en la mitad de las iteraciones del bucle más interno, y por tanto realiza

  $$\left(\sum_{j=i+1}^{n} (4+\frac{1}{2}9)+3 \right) + 3 = \frac{23}{2}(n-i)+3$$

  OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

  $$T(n) = \left(\sum_{i=1}^{n-1} \left(\frac{23}{2}(n-i)+3 \right) + 3 \right) + 3 = \frac{23}{4}n^2 + \frac{1}{4}n - 3 .$$

b) Como los tiempos de ejecución en los tres casos son polinomios de grado 2, la complejidad del algoritmo es cuadrática, independientemente de qué caso se trate.

Obsérvese cómo hemos analizado el tiempo de ejecución del algoritmo sólo en función de su código y no respecto a lo que hace, puesto que en muchos casos esto nos llevaría a conclusiones erróneas. Debe ser a posteriori cuando se analice el objetivo para el que fue diseñado el algoritmo.

En el caso que nos ocupa, un examen más detallado del código del procedimiento nos muestra que el algoritmo está diseñado para ordenar de forma creciente el vector que se le pasa como parámetro, siguiendo el método de la Burbuja. Lo que acabamos de ver es que sus casos mejor, peor y medio se producen respectivamente cuando el vector está inicialmente ordenado de forma creciente, decreciente y aleatoria.

Función *Algoritmo2* (👓)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2 OE (dos asignaciones).
- En la línea (2) se efectúa la condición del bucle, que supone 1 OE (la comparación).
- Las líneas (3) a (6) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2+2 y 2 OE respectivamente. Es importante hacer notar que el bucle también puede finalizar si se verifica la condición de la línea (4).
- Por último, la línea (9) supone 1 OE. A ella se llega cuando la condición del bucle *WHILE* deja de ser cierta.

Con esto:

- En el *caso mejor* se efectuarán solamente la líneas (1), (2), (3) y (4). En consecuencia, $T(n) = 2+1+3+3 = 9$.
- En el *caso peor* se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (9). Cada iteración del bucle está compuesta por las líneas (2) a (8), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se reducen a la mitad los elementos a considerar, por lo que el bucle se repite log$n$ veces. Por tanto,

$$T(n) = 2 + \left(\left(\sum_{i=1}^{\log n} (1+3+2+2+2) \right) + 1 \right) + 1 = 10 \log n + 4 .$$

- En el *caso medio*, necesitamos calcular el número medio de veces que se repite el bucle, y para esto veamos cuántas veces puede repetirse, y qué probabilidad tiene cada una de suceder.

  Por un lado, el bucle puede repetirse desde una vez hasta log$n$ veces, puesto que en cada iteración se divide por dos el número de elementos considerados. Si se repitiese una sola vez, es que el elemento ocuparía la posición $n/2$, lo que ocurre con una probabilidad $1/(n+1)$. Si el bucle se repitiese dos veces es que el elemento ocuparía alguna de las posiciones $n/4$ ó $3n/4$, lo cual ocurre con probabilidad $1/(n+1)+1/(n+1)=2/(n+1)$. En general, si se repitiese i veces es que el elemento ocuparía alguna de las posiciones $nk/2^i$, con k impar y $1 \le k < 2^i$.

  Es decir, el bucle se repite i veces con probabilidad $2^{i-1}/(n+1)$. Por tanto, el número medio de veces que se repite el ciclo vendrá dado por la expresión:

$$\sum_{i=1}^{\log n} i \frac{2^{i-1}}{n+1} = \frac{n \log n - n + 1}{n+1} .$$

  Con esto, la función ejecuta la línea (1) y después el bucle se repite ese número medio de veces, saliendo por la instrucción *RETURN* en la línea (4). Por consiguiente,

$$T(n) = 2 + \left(\frac{n \log n - n + 1}{n+1} \right)(1+3+2+2) + (1+3+3) = 9 + 8 \frac{n \log n - n + 1}{n+1}$$

c) En el caso mejor el tiempo de ejecución es una constante. Para los casos peor y medio, la complejidad resultante es de orden $\Theta(\log n)$ puesto que

$$\lim_{n \to \infty} \frac{T(n)}{\log n}$$

es una constante finita y distinta de cero en ambos casos (10 y 8 respectivamente).

Función *Euclides* (✍)

a) En este caso el análisis del tiempo de ejecución y la complejidad de la función sigue un proceso distinto al estudiado en los casos anteriores.

Lo primero es resaltar algunas características del algoritmo, siguiendo una línea de razonamiento similar a la de [BRA97]:

[1] Para cualquier par de enteros no negativos m y n tales que $n \geq m$, se verifica que n MOD $m < n/2$. Veámoslo:

a) Si $m > n/2$ entonces $1 \leq n/m < 2$ y por tanto n DIV $m = 1$, lo que implica que n MOD $m = n - m(n$ DIV $m) = n - m < n - n/2 = n/2$.

b) Por otro lado, si $m \leq n/2$ entonces n MOD $m < m \leq n/2$.

[2] Podemos suponer sin pérdida de generalidad que $n \geq m$. Si no, la primera iteración del bucle intercambia n con m ya que n MOD $m = n$ cuando $n < m$. Además, la condición $n \geq m$ se conserva siempre (es decir, es un invariante del bucle) pues n MOD m nunca es mayor que m.

[3] El cuerpo del bucle efectúa 4 OE, con lo cual el tiempo del algoritmo es del orden exacto del número de iteraciones que realiza el bucle. Por consiguiente, para determinar la complejidad del algoritmo es suficiente acotar este número.

[4] Una propiedad curiosa de este algoritmo es que no se produce un avance notable con cada iteración del bucle, sino que esto ocurre cada dos iteraciones. Consideremos lo que les ocurre a m y n cuando el ciclo se repite dos veces, suponiendo que no acaba antes. Sean m_0 y n_0 los valores originales de los parámetros, que podemos suponer $n_0 \geq m_0$ por [2]. Después de la primera iteración, m vale n_0 MOD m_0. Después de la segunda iteración, n toma ese valor, y por tanto ya es menor que $n_0/2$ (por [1]). En consecuencia, n vale menos de la mitad de lo que valía tras dos iteraciones del bucle. Como se sigue manteniendo que $n \geq m$, el mismo razonamiento se puede repetir para las siguientes dos iteraciones, y así sucesivamente.

El hecho de que n valga menos de la mitad cada dos iteraciones del bucle es el que nos permite intuir que el bucle se va a repetir del orden de $2\log n$ veces. Vamos a demostrar esto formalmente.

Para ello, vamos a tratar el bucle como si fuera un algoritmo recursivo. Sea $T(l)$ el número máximo de veces que se repite el bucle para valores iniciales m y n cuando $m \leq n \leq l$. En este caso l representa el tamaño de la entrada.

– Si $n \leq 2$ el bucle no se repite (si $m = 0$) o se hace una sola vez (si m es 1 ó 2).
– Si $n > 2$ y $m=1$ o bien m divide a n, el bucle se repite una sola vez.
– En otro caso ($n > 2$ y m no divide a n) el bucle se ejecuta dos veces, y por lo visto en [4], n vale a lo sumo la mitad de lo que valía inicialmente. En consecuencia $n \leq (l/2)$, y además m se sigue manteniendo por debajo de n.

Esto nos lleva a la ecuación en recurrencia $T(l) \leq 2 + T(l/2)$ si $l > 2$, $T(l) \leq 1$ si $l \leq 2$, lo que implica que el algoritmo de Euclides es de complejidad logarítmica respecto al tamaño de la entrada (l).

Nos preguntaremos la razón de usar $T(l)$ para acotar el número de iteraciones que realiza el algoritmo en vez de definir T directamente como una función de n, el mayor de los dos operandos, lo cual sería mucho más intuitivo.

El problema es que si definimos $T(n)$ como el número de iteraciones que realiza el algoritmo para los valores $m \le n$, no podríamos concluir que $T(n) \le 2 + T(n/2)$ del hecho de que n valga la mitad de su valor tras cada dos iteraciones del bucle.

Por ejemplo, para *Euclides(*8,13*)*, obtenemos que $T(13) = 5$ en el peor caso, mientras que $T(13/2) = T(6) = 2$. Esto ocurre porque tras dos iteraciones del bucle n no vale 6, sino 5 (y $m = 3$), y con esto sí es cierto que $T(13) \le 2 + T(5)$ ya que $T(5) = 3$.

La raíz de este problema es que esta nueva definición más intuitiva de T no lleva a una función monótona no decreciente ($T(5) > T(6)$) y por tanto la existencia de algún $n' \le n/2$ tal que $T(n) \le 2 + T(n')$ no implica necesariamente que $T(n) \le 2 + T(n/2)$.

En vez de esto, solamente podríamos afirmar que $T(n) \le 2 + max\{T(n') | n' \le n/2\}$, que es una ecuación en recurrencia bastante difícil de resolver. Esa es la razón de que escogiésemos nuestra función T de forma que fuera no decreciente y que expresara una cota superior del número de iteraciones.

Para acabar, es interesante hacer notar una característica curiosa de este algoritmo: se demuestra que su caso peor ocurre cuando m y n son dos términos consecutivos de la sucesión de Fibonacci.

b) $T(l) \in \Theta(\log l)$ como se deduce de la ecuación en recurencia que define el tiempo de ejecución del algoritmo.

Procedimiento *Misterio* (☺)

a) En este caso son tres bucles anidados los que se ejecutan, independientemente de los valores de la entrada, es decir, no existe peor, medio o mejor caso, sino un único caso.

Para calcular el tiempo de ejecución, veamos el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecuta 1 OE (una asignación).
– En la línea (2) se ejecutarán 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
– Igual ocurre con la línea (3), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
– Y también en la línea (4), esta vez con 2 OE (asignación y comparación) más las 2 adicionales de terminación del bucle.
– Por último, la línea (5) supone 2 OE (un incremento y una asignación).

Con esto, el bucle interno se ejecutará j veces, el medio $(n-i)$ veces, y el bucle exterior $(n-1)$ veces, lo que conlleva un tiempo de ejecución de:

$$T(n)=1+\left(\sum_{i=1}^{n-1}\left(3+\left(\sum_{j=i+1}^{n}\left(3+\left(\sum_{k=1}^{j}(2+2)\right)+2\right)\right)+3\right)\right)+3=$$

$$1+\left(\sum_{i=1}^{n-1}\left(3+\left(\sum_{j=i+1}^{n}\left(3+\left(4j\right)+2\right)\right)+3\right)\right)+3=1+\left(\sum_{i=1}^{n-1}\left(3+\left(\sum_{j=i+1}^{n}\left(4j+5\right)\right)+3\right)\right)+3=$$

$$1+\left(\sum_{i=1}^{n-1}\left(3+\left(2(n+i)+7\right)\left(n-i\right)+3\right)\right)+3=$$

$$1+\frac{8n^3+15n^2+13n-36}{6}+3=\frac{4}{3}n^3+\frac{15}{6}n^2+\frac{13}{6}n-2\,.$$

b) Como el tiempo de ejecución es un polinomio de grado 3, la complejidad del algoritmo es de orden $\Theta(n^3)$.

Solución al Problema 1.6 (☺)

Para comprobar que $O(f) \subset O(g)$ en cada caso y que esa inclusión es estricta, basta ver que $lim_{n\to\infty}(f(n)/g(n)) = 0$, pues todas las funciones son continuas y por tanto los límites existen. Por consiguiente,

$O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!)$.

Solución al Problema 1.7 (☺)

a) Por la definición de O, sabemos que $f\in O(g)$ si y sólo si existen $c_1 > 0$ y n_1 tales que $f(n) \leq c_1 g(n)$ para todo $n \geq n_1$.

Análogamente, por la definición de Ω tenemos que $g\in\Omega(f)$ si y sólo si existen $c_2 > 0$ y n_2 tales que $g(n) \geq c_2 f(n)$ para todo $n \geq n_2$. Por consiguiente,

$\Rightarrow$) Si $f\in O(g)$ basta tomar $c_2=1/c_1$ y $n_2=n_1$ para ver que $g(n)\in\Omega(f)$.

$\Leftarrow$) Recíprocamente, si $g\in\Omega(f)$ basta tomar $c_1=1/c_2$ y $n_1=n_2$ para que $f\in O(g)$.

Obsérvese que esto es posible pues c_1 y c_2 son ambos estrictamente mayores que cero, y por tanto poseen inverso.

b) Sean $f(n)=\begin{cases} n^2 & \text{si } n \text{ es par.} \\ 1 & \text{si } n \text{ es impar.}\end{cases}$ y $g(n)=n^2$.

Entonces $\Theta(g)=\Theta(n^2)$, y por otro lado $O(f)=O(n^2)$, con lo cual $f\in O(n^2)=O(g)$. Sin embargo, si n es impar no puede existir $c>0$ tal que $f(n) = 1 \geq cn^2 = cg(n)$, y por consiguiente $f\notin\Omega(g)$.

Intuitivamente, lo que buscamos es una función f cuyo crecimiento asintótico estuviera acotado superiormente por g (es decir, que f no creciera "más deprisa" que g) y que sin embargo f no estuviera acotado inferiormente por g.

c) Veamos que $\log_a n \in \Theta(\log_b n)$.

Sabemos por las propiedades de los logaritmos que si a y b son números reales mayores que 1 se cumple que

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

Con esto,

$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \log_a b = \log_a b,$$

que es una constante real finita distinta de cero (pues $a,b>1$), y por tanto $\Theta(\log_a n) = \Theta(\log_b n)$.

Solución al Problema 1.8 (☺)

Para justificar estas afirmaciones nos apoyaremos en la definición de O y Ω, y trataremos de encontrar la constante real c y el número natural n_0 que caracterizan las inecuaciones que definen a ambas cotas.

– $f_1 \in O(f_2)$ pues $n^2 \le n^2 + 1000n$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).

– $f_1 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 \le cn$.

– $f_1 \in O(f_4)$ pues $n^2 \le n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

– $f_2 \in O(f_1)$ pues basta tomar c y n_0 tales que $c > 1 + 1000/n$ para todo $n \ge n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.

– $f_2 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 + 1000n \le cn$.

– $f_2 \in O(f_4)$ pues $n^2 + 1000n \le n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

– $f_3 \notin O(f_1)$ pues si n es par no existen c y n_0 tales que $n^3 \le cn^2$.

– $f_3 \notin O(f_2)$ pues si n es par no existen c y n_0 tales que $n^3 \le c(n^2 + 1000n)$.

– $f_3 \in O(f_4)$ pues $f_3(n) \le n^3 = f_4(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

– $f_4 \notin O(f_1)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \le cn^2$.

– $f_4 \notin O(f_2)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \le c(n^2 + 1000n)$.

– $f_4 \notin O(f_3)$ pues si $n > 100$, n impar, no existen c y n_0 tales que $n^3 \le cn$.

– $f_1 \in \Omega(f_2)$ pues $n^2 \ge c(n^2 + 1000n)$ para c y n_0 tales que $c > 1 + 1000/n$ para todo $n \ge n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.

– $f_1 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 \ge cn^3$.

– $f_1 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 \geq cn^3$.
– $f_2 \in \Omega(f_1)$ pues $n^2 + 100n \geq n^2$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).
– $f_2 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
– $f_2 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
– $f_3 \notin \Omega(f_1)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^2$.
– $f_3 \notin \Omega(f_2)$ pues si n es impar no existen c y n_0 tales que $n \geq c(n^2 + 1000n)$.
– $f_3 \notin \Omega(f_4)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^3$.
– $f_4 \in \Omega(f_1)$ pues $n^3 \geq n^2$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
– $f_4 \in \Omega(f_2)$ pues $n^3 \geq n^2 + 1000n$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
– $f_4 \in \Omega(f_3)$ pues $n^3 \geq f_3(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

Obsérvese que $\Theta(f_1) = \Theta(f_2) = \Theta(n^2)$, $\Theta(f_4) = \Theta(n^3)$, $\Omega(f_3) = \Omega(n)$, y $O(f_3) = O(n^3)$.

Solución al Problema 1.9 (☺/☹)

Para resolver estas ecuaciones seguiremos generalmente el mismo método, basado en los resultados expuestos al comienzo del capítulo. Primero intentaremos transformar la ecuación en recurrencia en una ecuación de la forma:

$$a_0T(n) + a_1T(n-1) + ... + a_kT(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + ... + b_s^n p_s(n),$$

para después resolver su ecuación característica asociada. Con las raíces de esta ecuación es fácil ya obtener el término general de la función buscada.

a) $T(n) = 3T(n-1) + 4T(n-2)$ si $n>1$; $T(0) = 0$; $T(1) = 1$.

Escribiendo la ecuación de otra forma:

$$T(n) - 3T(n-1) - 4T(n-2) = 0,$$

ecuación en recurrencia homogénea con ecuación característica $x^2-3x-4 = 0$. Resolviendo esta ecuación, sus raíces son 4 y –1, con lo cual:

$$T(n) = c_1 4^n + c_2(-1)^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{array}{l} 0 = T(0) = c_1 4^0 + c_2(-1)^0 = c_1 + c_2 \\ 1 = T(1) = c_1 4^1 + c_2(-1)^1 = 4c_1 - c_2 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 1/5 \\ c_2 = -1/5 \end{array}\right\}$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{5}\left(4^n - (-1)^n\right) \in O(4^n).$$

b) $T(n) = 2T(n-1) - (n+5)3^n$ si $n>0$; $T(0) = 0$.

Reescribiendo la ecuación obtenemos:

$$T(n) - 2T(n-1) = -(n+5)3^n,$$

que es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-3)^2 = 0$, de raíces 2 y 3 (esta última con grado de multiplicidad dos), con lo cual

$$T(n) = c_1 2^n + c_2 3^n + c_3 n 3^n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$T(1) = 2T(0) - 6\cdot 3 = -18$$
$$T(2) = 2T(1) - 7\cdot 9 = -99$$

Con esto:

$$\left.\begin{array}{l} 0 = T(0) = c_1 2^0 + c_2 3^0 + c_3 0\cdot 3^0 = c_1 + c_2 \\ -18 = T(1) = c_1 2^1 + c_2 3^1 + c_3 1\cdot 3^1 = 2c_1 + 3c_2 + 3c_3 \\ -99 = T(2) = c_1 2^2 + c_2 3^2 + c_3 2\cdot 3^2 = 4c_1 + 9c_2 + 18c_3 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 9 \\ c_2 = -9 \\ c_3 = -3 \end{array}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = 9\cdot 2^n - 9\cdot 3^n - 3n3^n \in \Theta(n3^n).$$

Existe otra forma de resolver este tipo de problemas, que se basa en manipular la ecuación original hasta convertirla en homogénea. Partiendo de la ecuación $T(n) - 2T(n-1) = (n+5)3^n$, necesitamos escribir un sistema de ecuaciones basado en ella que permita anular el término no dependiente de $T(n)$. Para ello:

– primero escribimos la recurrencia original,
– la segunda ecuación se obtiene reemplazando n por $n-1$ y multiplicando por -6,
– y la tercera se obtiene reemplazando n por $n-2$ y multiplicando por 9.

De esta forma obtenemos:

$$\begin{array}{llll} T(n) - 2T(n-1) & & & = (n+5)3^n \\ \quad - 6T(n-1) & +12T(n-2) & & = -6(n+4)3^{n-1} \\ & 9T(n-2) & -18T(n-3) & = 9(n+3)3^{n-2} \end{array}$$

Sumando estas tres ecuaciones conseguimos una ecuación homogénea:

$$T(n) - 8T(n-1) + 21T(n-2) - 18T(n-3) = 0$$

cuya ecuación característica es $x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$. A partir de aquí se puede resolver mediante el proceso descrito anteriormente.

Como puede observarse, este método es más intuitivo pero menos metódico y ordenado que el que hemos utilizado para solucionar ecuaciones en recurrencia. Además, no hay una única forma de plantear esta ecuaciones.

c) $T(n) = 4T(n/2) + n^2$ si $n>4$, n potencia de 2; $T(1) = 1$; $T(2) = 8$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 4T(2^{k-1}) + 2^{2k}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 4^k,$$

ecuación no homogénea con ecuación característica $(x-4)^2 = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 k 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 4^k + c_2 k 4^k = c_1 2^{2k} + c_2 k 2^{2k}$$

y después $n = 2^k$, obteniendo finalmente

$$T(n) = c_1 n^2 + c_2 n^2 \log n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{array}{l} 1 = T(1) = c_1 1^2 + c_2 1^2 \cdot 0 = c_1 \\ 8 = T(2) = c_1 2^2 + c_2 2^2 \cdot 1 = 4c_1 + 4c_2 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 1 \\ c_2 = 1 \end{array}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = n^2 + n^2 \log n \in \Theta(n^2 \log n).$$

Existe otra forma de resolver este tipo de problemas, mediante el desarrollo "telescópico" de la ecuación en recurrencia. Escribiremos la ecuación hasta llegar a una expresión en donde sólo aparezcan las condiciones iniciales:

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2 = 4\left(4T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right) + n^2 =$$

$$= 4^2 T\left(\frac{n}{4}\right) + 2n^2 = 4^2\left(4T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right) + 2n^2 =$$

$$= 4^3 T\left(\frac{n}{8}\right) + 3n^2 = ... = 4^x T(1) + xn^2.$$

De esta forma hemos ido desarrollando los términos de esta sucesión, cada uno en función de términos anteriores. Sólo nos queda por calcular el número de términos (x) que hemos tenido que desarrollar.

Pero ese número x coincide con el número de términos de la sucesión $n/2$, $n/4$, $n/8$,...,4,2,1, que es $\log n$ pues n es una potencia de 2. En consecuencia,

$$T(n) = 4^{\log n}\, T(1) + \log n \cdot n^2 = n^{\log 4} \cdot 1 + \log n \cdot n^2 = n^2 + \log n \cdot n^2 \in \Theta(n^2 \log n).$$

d) $T(n) = 2T(n/2) + n\log n$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k2^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-2)^3 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k 2^k + c_3 k^2 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 k 2^k + c_3 k^2 2^k,$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 n + c_2 n\log n + c_3 n\log^2 n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$n\log n = T(n) - 2T(n/2) = (c_3 - c_2)n + 2c_3 n\log n,$$

por lo que $c_3 = c_2$ y $2c_3 = 1$, de donde

$$T(n) = c_1 n + 1/2 n\log n + 1/2 n\log^2 n.$$

En consecuencia $T(n) \in \Theta(n\log^2 n)$ independientemente de las condiciones iniciales.

e) $T(n) = 3T(n/2) + 5n + 3$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 3T(2^{k-1}) + 5 \cdot 2^k + 3.$$

Llamando $t_k = T(2^k)$, la ecuación final es:

$$t_k = 3t_{k-1} + 5\cdot 2^k + 3,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-3)(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 3^k + c_2 2^k + c_3.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 3^k + c_2 2^k + c_3$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 3^{\log n} + c_2 n + c_3 = c_1 n^{\log 3} + c_2 n + c_3.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log 3} + c_2 n + c_3 = 3(c_1(n^{\log 3}/3) + c_2 n/2 + c_3) + 5n + 3.$$

Igualando los coeficientes de $n^{\log 3}$, n y los términos independientes obtenemos $c_3=-3/2$ y $c_2=-10$, de donde

$$T(n) = c_1 n^{\log 3} - 10n - 3/2.$$

Como $\log 3 > 1$, $T(n)$ será de complejidad $\Theta(n^{\log 3})$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 3T(1) + 10 + 3.$$

Por otro lado, basándonos en la ecuación que hemos obtenido,

$$T(2) = 3c_1 - 20 - 3/2.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = T(1) + 23/2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(n^{\log 3}) & \text{si } T(1) \neq -23/2 \\ \Theta(n) & \text{si } T(1) = -23/2 \end{cases}$$

f) $T(n) = 2T(n/2) + \log n$ si $n>1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 2t_{k-1} = k$$

y cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 + c_3 k$$

y después $n = 2^k$ ($k = \log n$), y por tanto

$$T(n) = c_1 n + c_2 + c_3 \log n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n + c_2 + c_3 \log n = 2(c_1 n/2 + c_2 + c_3 \log n - c_3) + \log n.$$

Igualando los coeficientes de $\log n$ y los términos independientes obtenemos que $c_3 = -1$ y $c_2 = -2$, de donde

$$T(n) = c_1 n - 2 - \log n.$$

Esta función será de orden de complejidad $\Theta(n)$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(\log n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 2T(1) + 1.$$

Por otro lado, basándonos en la ecuación que hemos obtenido

$$T(2) = 2c_1 - 2 - 1.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = \mathrm{T}(1) + 2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(\log n) & \text{si } T(1) = -2 \\ \Theta(n) & \text{si } T(1) \neq -2 \end{cases}$$

g) $T(n) = 2T(n^{1/2}) + \log n$ con $n = 2^{2^k}$; $T(2) = 1$.

Haciendo el cambio $n = 2^{2^k}$ ($k = \log\log n$) obtenemos la ecuación

$$T(2^{2^k}) = 2T(2^{2^{k-1}}) + \log 2^{2^k}.$$

Llamando $t_k = T(2^{2^k})$, la ecuación final es

$$t_k = 2t_{k-1} + 2^k,$$

ecuación en recurrencia no homogénea cuya ecuación característica es $(x-2)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^{2^k})$, con lo que

$$T(2^{2^k}) = c_1 2^k + c_2 k 2^k$$

y después $n = 2^{2^k}$ (k = loglog$n$, o bien log$n = 2^k$), por lo cual tenemos que

$$T(n) = c_1 \log n + c_2 \log n \cdot \log\log n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como disponemos de sólo una y tenemos dos incógnitas, usamos la ecuación original para obtener la otra:

$$T(4) = 2T(2) + \log 4 = 4.$$

Con esto:

$$\left.\begin{array}{l} 1 = T(2) = c_1 \log 2 + c_2 \log 2 \cdot 0 = c_1 \\ 4 = T(4) = c_1 \log 4 + c_2 \log 4 \cdot \log\log 4 = 2c_1 + 2c_2 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 1 \\ c_2 = 1 \end{array}\right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = \log n + \log n \cdot \log\log n \in \Theta(\log n \cdot \log\log n).$$

h) $T(n) = 5T(n/2) + (n\log n)^2$ si $n>1$, n potencia de 2; $T(1)=1$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, k = log$n$) obtenemos

$$T(2^k) = 5T(2^{k-1}) + (k2^k)^2 = 5T(2^{k-1}) + k^2 4^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 5t_{k-1} + k^2 4^k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 5\, t_{k-1} = k^2 4^k,$$

cuya ecuación característica asociada es $(x-5)(x-4)^3 = 0$. Por tanto,

$$t_k = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k$$

y después $n = 2^k$ ($k = \log n$), por tanto

$$\begin{aligned} T(n) = {} & c_1 5^{\log n} + c_2 4^{\log n} + c_3 \log n 4^{\log n} + c_4 \log^2 n 4^{\log n} = \\ & c_1 n^{\log 5} + c_2 n^{\log 4} + c_3 \log n \cdot n^{\log 4} + c_4 \log^2 n \cdot n^{\log 4} = \\ & c_1 n^{\log 5} + c_2 n^2 + c_3 n^2 \log n + c_4 n^2 \log^2 n. \end{aligned}$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$\begin{aligned} & T(2) = 5T(1) + 2^2 = 9; \\ & T(4) = 5T(2) + 8^2 = 109; \\ & T(8) = 5T(4) + 24^2 = 1121. \end{aligned}$$

Con esto:

$$\left.\begin{aligned} & 1 = T(1) = c_1 1 + c_2 1 + c_3 1 \cdot 0 + c_4 1 \cdot 0 = c_1 + c_2 \\ & 9 = T(2) = c_1 5 + c_2 4 + c_3 4 \cdot 1 + c_4 4 \cdot 1 = 5c_1 + 4c_2 + 4c_3 + 4c_4 \\ & 109 = T(4) = c_1 25 + c_2 16 + c_3 16 \cdot 2 + c_4 16 \cdot 4 = 25c_1 + 16c_2 + 32c_3 + 64c_4 \\ & 1121 = T(8) = c_1 125 + c_2 64 + c_3 64 \cdot 3 + c_4 64 \cdot 9 = 125c_1 + 64c_2 + 192c_3 + 576c_4 \end{aligned}\right\}$$

Solucionando el sistema de ecuaciones obtenemos los valores de las constantes:

$$\left.\begin{aligned} c_1 &= 181 \\ c_2 &= -180 \\ c_3 &= -40 \\ c_4 &= -4 \end{aligned}\right\}$$

y sustituyéndolos en la ecuación anterior, obtenemos

$$T(n) = 181n^{\log 5} - 180n^2 - 40n^2 \log n - 4n^2 \log^2 n \in \Theta(n^2 \log^2 n).$$

i) $T(n) = T(n-1) + 2T(n-2) - 2T(n-3)$ si $n>2$; $T(n) = 9n^2 - 15n + 106$ si n=0,1,2.

Reescribiendo la ecuación:

$$T(n) - T(n-1) - 2T(n-2) + 2T(n-3) = 0,$$

ecuación en recurrencia homogénea cuya ecuación característica asociada es

$$x^3 - x^2 - 2x + 2 = 0.$$

Resolviendo esa ecuación, sus raíces son 1, $\sqrt{2}$ y $-\sqrt{2}$, con lo cual

$$T(n) = c_1 + c_2(\sqrt{2})^n + c_3(-\sqrt{2})^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left.\begin{array}{l} 106 = T(0) = c_1 + c_2 + c_3 \\ 100 = T(1) = c_1 + c_2\sqrt{2} - c_3\sqrt{2} \\ 112 = T(2) = c_1 + 2c_2 + 2c_3 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 100 \\ c_2 = 3 \\ c_3 = 3 \end{array}\right\} \Rightarrow T(n) = 100 + 3\sqrt{2^n}\left(1 + (-1)^n\right)$$

.

En consecuencia, $T(n) \in \Theta(2^{n/2})$.

j) $T(n) = (3/2)T(n/2)–(1/2)T(n/4)–(1/n)$ si $n>2$, n potencia de 2; $T(1)=1$; $T(2)=3/2$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = (3/2)T(2^{k-1}) - (1/2)T(2^{k-2}) - (1/2)^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = (3/2)t_{k-1} - (1/2)t_{k-2} - (1/2)^k$$

ecuación en recurrencia no homogénea en la forma

$$t_k - (3/2)t_{k-1} + (1/2)t_{k-2} = -(1/2)^k,$$

cuya ecuación característica asociada es $(x-1)(x-1/2)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2 2^{-k} + c_3 k 2^{-k}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 + c_2 2^{-k} + c_3 k 2^{-k}$$

después $n = 2^k$ ($k = \log n$), con lo cual

$$T(n) = c_1 + (1/n)c_2 + c_3(\log n/n).$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de dos y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener la tercera:

$$T(4) = (3/2)T(2) - (1/2)T(1) - (1/4) = 3/2.$$

Con esto:

$$\left.\begin{array}{l} 1 = T(1) = c_1 + c_2 \\ 3/2 = T(2) = c_1 + c_2(1/2) + c_3(1/2) \\ 3/2 = T(4) = c_1 + c_2(1/4) + c_3(1/2) \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 1 \\ c_2 = 0 \\ c_3 = 1 \end{array}\right\} \Rightarrow T(n) = 1 + \frac{\log n}{n} \in \Theta(1).$$

k) $T(n) = 2T(n/4) + n^{1/2}$ si $n>4$, n potencia de 4.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-2}) + 2^{k/2}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-2} + 2^{k/2},$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 2t_{k-2} = (\sqrt{2}\,)^k$$

cuya ecuación característica asociada es $(x^2-2)(x-\sqrt{2}\,) = 0$, o lo que es igual, $(x+\sqrt{2}\,)(x-\sqrt{2}\,)^2 = 0$. Por tanto,

$$t_k = c_1(-\sqrt{2}\,)^k + c_2(\sqrt{2}\,)^k + c_3k(\sqrt{2}\,)^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1(-\sqrt{2}\,)^k + c_2(\sqrt{2}\,)^k + c_3k(\sqrt{2}\,)^k$$

después $n = 2^k$ ($k = \log n$), y por tanto obtenemos:

$$T(n) = \sqrt{n}\ (c_1(-1)^{\log n} + c_2 + c_3\log n).$$

Si n es múltiplo de 4 entonces $\log n$ es par, y por tanto $(-1)^{\log n}$ vale siempre 1. Esto nos permite afirmar, llamando $c_0 = c_1 + c_2$, que

$$T(n) = \sqrt{n}\ (c_0 + c_3\log n).$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$\sqrt{n}\ (c_0 + c_3\log n) = 2(\sqrt{n}/2\ (c_0 + c_3\log n - 2c_3)) + \sqrt{n}\ .$$

Igualando los coeficientes de $\sqrt{n}$, $\sqrt{n}\log n$ y los términos independientes obtenemos $c_3=1/2$, de donde

$$T(n) = \sqrt{n}\ (c_0 + 1/2\log n) \in \Theta(\sqrt{n}\log n).$$

Este problema también podría haberse solucionado mediante otro cambio, $n = 4^k$ (o, lo que es igual, $k = \log_4 n$) obteniendo la ecuación

$$T(4^k) = 2T(4^{k-1}) + 4^{k/2}.$$

Esta nos lleva, tras llamar $t_k = T(4^k)$, a la ecuación final

$$t_k = 2t_{k-1} + 2^k,$$

cuya resolución conduce a la misma solución que la obtenida mediante el primer cambio.

l) $T(n) = 4T(n/3) + n^2$ si $n>3$, n potencia de 3.

Haciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$) obtenemos que

$$T(3^k) = 4T(3^{k-1}) + 9^k.$$

Llamando $t_k = T(3^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 9^k,$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 4t_{k-1} = 9^k,$$

cuya ecuación característica asociada es $(x-4)(x-9) = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 9^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(3^k)$, con lo que

$$T(3^k) = c_1 4^k + c_2 3^{2k}$$

y después $n = 3^k$ ($k = \log_3 n$), y por tanto

$$T(n) = c_1 4^{\log_3 n} + c_2 n^2 = c_1 n^{\log_3 4} + c_2 n^2.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log_3 4} + c_2 n^2 = 4\left(c_1\left(\frac{n}{3}\right)^{\log_3 4} + c_2\left(\frac{n}{3}\right)^2\right) + n^2 = c_1 n^{\log_3 4} + \left(\frac{4}{9}c_2 + 1\right)n^2$$

Igualando los coeficientes de $n^{\log_3 4}$ y de n^2 obtenemos $c_2 = 9/5$, de donde

$$T(n) = c_1 n^{\log_3 4} + \frac{9}{5}n^2.$$

Como $\log_3 4 < 2$, entonces $T(n) \in \Theta(n^2)$.

Solución al Problema 1.10 (☺)

a) Cierto. Se deduce de la propiedad 6 del apartado 1.3.1, pero veamos una posible demostración directa:

Si $T_1 \in O(f)$, sabemos que existen $c_1 > 0$ y n_1 tales que $T_1(n) \le c_1 f(n)$ para todo $n \ge n_1$. Análogamente, como $T_2 \in O(f)$, existen $c_2 > 0$ y n_2 tales que $T_2(n) \le c_2 f(n)$ para $n \ge n_2$. [1.1]

Para comprobar que $T_1 + T_2 \in O(f)$, debemos encontrar una constante real $c > 0$ y un número natural n_0 tales que $T_1(n) + T_2(n) \le cf(n)$ para todo $n \ge n_0$. [1.2]

Apoyándonos en [1.1], basta tomar $n_0 = máx\{n_1, n_2\}$ y $c = c_1 + c_2$, con las que se verifica la ecuación [1.2] para todo $n \ge n_0$.

Existe otra forma de demostrarlo, utilizando límites en caso de que estos existan, como sucede por ejemplo cuando las funciones son continuas:

Si $T_1 \in O(f)$, entonces $\lim_{n\to\infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$.

Análogamente, como $T_2 \in O(f)$, $\lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.3]

Veamos entonces que $\lim_{n\to\infty} \frac{T_1(n)+T_2(n)}{f(n)} = k < \infty$. [1.4]

Pero [1.4] es cierto pues, como los dos límites en [1.3] son finitos y positivos podemos conmutar la suma con el límite y obtenemos que

$$\lim_{n\to\infty} \frac{T_1(n)+T_2(n)}{f(n)} = \lim_{n\to\infty} \frac{T_1(n)}{f(n)} + \lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_1 + k_2 < \infty.$$

b) Cierto.

Análogamente a lo realizado en el apartado anterior, si $T_1 \in O(f)$, entonces $\lim_{n\to\infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$. Igualmente, como $T_2 \in O(f)$, $\lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.5]

Veamos entonces que $\lim_{n\to\infty} \frac{T_1(n)-T_2(n)}{f(n)} = k < \infty$. [1.6]

Pero [1.6] es cierto pues, como los dos límites en [1.5] existen y son finitos y positivos podemos conmutar la resta con el límite y obtenemos que

$$\lim_{n\to\infty} \frac{T_1(n)-T_2(n)}{f(n)} = \lim_{n\to\infty} \frac{T_1(n)}{f(n)} - \lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_1 - k_2 < \infty.$$

c) Falso.

Consideremos $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1(n)/T_2(n) = n \notin O(1)$.

d) Falso.

Consideremos de nuevo $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1 \notin O(T_2)$ pues $n^2 \notin O(n)$.

Solución al Problema 1.11 (☺)

Sean $f(n) = n$ y $g(n) = \begin{cases} n^2, & \text{si } n \text{ es par.} \\ 1, & \text{si } n \text{ es impar.} \end{cases}$

Si n es impar, no podemos encontrar ninguna constante c tal que

$$f(n) = n \leq cg(n) = c,$$

y por tanto $f \notin O(g)$. Por otro lado, si n es par no podemos encontrar ninguna constante c tal que

$$g(n) = n^2 \leq cf(n) = cn,$$

y por tanto $g \notin O(f)$.

Solución al Problema 1.12 (☺)

Para comprobar que $\log^k n \in O(n)$ basta ver que

$$\lim_{n\to\infty} \frac{\log^k n}{n} = 0$$

para todo k. Pero eso es cierto siempre. Obsérvese además que por esa misma razón $\log^k n \notin \Omega(n)$ para cualquier $k > 0$.

Solución al Problema 1.13

Vamos a suponer que los tiempos de ejecución de las funciones *Esvacio*, *Izq*, *Der* y *Raiz* es de c operaciones elementales (OE), que el tiempo de ejecución de *Opera* es d OE, y el de *Max2* es 1 OE.

Procedimiento *Inorden* (☺)

Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan $2+c$ OE: la llamada a *Esvacio* (1 OE), el tiempo de ejecución de este procedimiento (c) y una negación.
– En la línea (2) se efectúa la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Inorden* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq(t)*.
– En la línea (3) se ejecutan $2+c+d$ OE: dos llamadas a procedimientos y sus respectivos tiempos de ejecución.
– El número de OE de la línea (4) se calcula de forma análoga a la línea (2): $2+c$ más lo que tarda *Inorden* en ejecutarse con el número de elementos que hay en *Der(t)*.

Para estudiar el tiempo de ejecución, vamos a considerar dos casos extremos: que el árbol sea degenerado (es decir, una lista) y que sea equilibrado. Cualquier árbol se encuentra en una situación intermedia a estos dos casos.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t))* y que para todo a subárbol de t se verifica que *Esvacio(Izq(a))*. Por tanto, el número de OE que se realizan en la ejecución de *Inorden(t)* para un árbol t con n elementos es:

$$T(n) = (2+c) + (2+c+T(0)) + (2+c+d) + (2+c+T(n-1)) = 8+4c+d+T(0)+T(n-1).$$
$$T(0) = 2+c.$$

  Con esto, $T(n) = 10 + 5c + d + T(n-1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

$$T(n) = 10 + 5c + d + T(n-1) = (10 + 5c + d) + (10 + 5c + d) + T(n-2) = ... = (10 + 5c + d)n + (2+c) \in \Theta(n)$$

- Si t es equilibrado sus dos subárboles (izquierdo y derecho) tienen del orden de $n/2$ elementos y son a su vez equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Inorden(t)* para un árbol t con n elementos es:

$$T(n)= (2+c) + (2+c+T(n/2)) + (2+c+d) + (2+c+T(n/2)) = 8+4c+d+2T(n/2).$$
$$T(0)= 2+c.$$

  Para resolver esta ecuación en recurrencia se hace el cambio $t_k = T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 4c + d,$$

  ecuación no homogénea con ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2$$

  y, deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0)=2+c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 4c + d + 2(2 + c)$, obteniendo

$$T(n) = (10 + 5c + d)n + (2+c) \in \Theta(n).$$

Función *Altura* (☺)

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan $1+c$ OE: la llamada a *Esvacio* (1 OE) más el tiempo de ejecución de este procedimiento (c OE).
– En la línea (2) se realiza 1 OE.
– En la línea (4) se efectúan:
  a) la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq(t)*; más
  b) la llamada a *Der* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Der(t)*; más
  c) el cálculo del máximo de ambos números (1 OE), un incremento (1 OE) y el *RETURN* (1 OE).

Para estudiar el tiempo de ejecución de esta función consideraremos los mismos casos que para la función *Inorden*: que el árbol sea degenerado (es decir, una lista) o que sea equilibrado.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t))* y que para todo a subárbol de t se verifica que *Esvacio(Izq(a))*. Por tanto, el número de OE que se realizan en la ejecución de *Altura(t)* para un árbol t con n elementos es:

  $$T(n)= (1+c) + (1+c+1+T(0) + 1+c+1+T(n-1)) +3 = 8 + 3c + T(0) + T(n-1).$$
  $$T(0)= (1+c) + 1 = 2+c.$$

  Con esto, $T(n)=10+4c+T(n-1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

  $$T(n) = 10 + 4c + T(n-1) = (10 + 4c) + (10 + 4c) + T(n-2) = ... = (10 + 4c)n + (2 + c) \in \Theta(n)$$

- Si t es equilibrado sus dos subárboles tienen del orden de $n/2$ elementos y son también equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Altura(t)* para un árbol t con n elementos es:

  $$T(n)= (1+c) + (1+c+1+T(n/2)) + 1+c+1+T(n/2)) +3 = 8 + 3c + 2T(n/2).$$
  $$T(0)= 2+c.$$

Para resolver esta ecuación en recurrencia se hace el cambio $t_k=T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 3c,$$

ecuación no homogénea de ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2.$$

Deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0)=2+c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 3c + 2(2 + c)$. Finalmente obtenemos

$$T(n) = (10 + 4c)n + (2 + c) \in \Theta(n).$$

Función *Mezcla* (☞)

Para resolver este problema vamos a suponer que el tiempo de ejecución del procedimiento *Ins*, que inserta un elemento en un árbol binario de búsqueda, es $A\log n+B$, siendo A y B dos constantes. Supongamos también que n y m son el número de elementos de *t1* y *t2* respectivamente.

Para estudiar el tiempo de ejecución $T(n,m)$ consideraremos, al igual que hicimos para la función anterior, dos casos extremos: que el árbol *t2* sea degenerado (es decir, una lista) o que sea equilibrado.

- Si *t2* es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio(Izq(t2))* y que para todo *a* subárbol de *t2* se verifica que *Esvacio(Izq(a))*. Por tanto, vamos a ver el número de OE que se realizan en cada línea de la función en este caso:
  - En la línea (1) se invoca a *Esvacio(t1)*, lo que supone $1+c$ OE.
  - En la línea (2) se efectúa 1 OE.
  - Análogamente, las líneas (3) y (4) realizan $(1+c)$ y 1 respectivamente.
  - Para estudiar el número de OE que realiza la línea (6), vamos a dividirla en cuatro partes:
    a) *a1:=Ins(t1,Raiz(t2))*, siendo *a1* una variable auxiliar para efectuar los cálculos. Se efectúan $2+c+A\log n+B$ operaciones elementales: la llamada a *Raiz* (1), el tiempo que ésta tarda (c), la llamada a *Ins* (1 OE), y su tiempo de ejecución ($A\log n+B$).
    b) *a2:=Mezcla(a1,Izq(t2))*, siendo *a2* una variable auxiliar para efectuar los cálculos. Se efectúan aquí $2+c+T(n+1,0)$ operaciones elementales: llamada a I*zq* (1), el tiempo que ésta tarda (c), la llamada a M*ezcla* (1 OE), y su tiempo de ejecución, que será $T(n+1,0)$, pues estamos suponiendo que *Esvacio(Izq(a))* para todo *a* subárbol de *t2*.
    c) *a3:=Mezcla(a2,Der(t2))*, siendo *a3* una variable auxiliar para efectuar los cálculos. Se efectúan $2+c+T(n+1,m-1)$ operaciones elementales: la

llamada a *Der* (1), el tiempo que ésta tarda (c), la llamada a *Mezcla* (1 OE), y su tiempo de ejecución, que será $T(n+1,m-1)$, pues estamos suponiendo que *Esvacio(Izq(a))* para todo a subárbol de *t2* o, lo que es igual, que el número de elementos de *Der(t)* es $m-1$.

d) *RETURN a3*, que realiza 1 OE.

Por tanto, la ejecución de *Mezcla(t1,t2)* en este caso es :

$$T(n,m) = 9 + 5c + B + A\log n + T(n+1,0) + T(n+1,m-1)$$

con las condiciones iniciales $T(0,m) = 2 + c$ y $T(n,0) = 3 + 2c$. Para resolver la ecuación en recurrencia podemos expresarla como:

$$T(n,m) = 12 + 7c + B + A\log n + T(n+1,m-1)$$

haciendo uso de la segunda condición inicial. Desarrollando telescópicamente la ecuación:

$$\begin{aligned} T(n,m) &= 12 + 7c + B + A\log n + T(n+1,m-1) = \\ &= (12+7c+B+A\log n) + (12+7c+B+A\log(n+1)) + T(n+2,m-2) = \\ &\ldots\ldots\ldots\ldots\ldots \\ &= m(12+7c+B) + \left(\sum_{i=0}^{m-1} A\log(n+i)\right) + T(n+m,0) = \\ &= m(12+7c+B) + 2c + 3 + A\left(\sum_{i=0}^{m-1}\log(n+i)\right). \end{aligned}$$

Pero como $\log(n+i) \le \log(n+m)$ para todo $0 \le i \le m$,

$$T(n,m) \le m(12 + 7c + B) + 2c + 3 + Am\log(n+m) \in O(m\log(n+m))$$

- El segundo caso es que *t2* sea equilibrado, para el que se demuestra de forma análoga que

$$T(n,m) \in O(m\log(n+m)).$$

Solución al Problema 1.14 (☺)

Para comprobar que $O(f)\subset O(g)$, basta ver que $lim_{n\to\infty} (f(n)/g(n)) = 0$ en cada caso pues las funciones son continuas, lo que implica la existencia de los límites. De esta forma se obtiene la siguiente ordenación:

$$O((1/3)^n) \subset O(17) \subset O(\log\log n) \subset O(\log n) \subset O(\log^2 n) \subset O(\sqrt{n}) \subset O(\sqrt{n}\log^2 n)$$
$$\subset O(n/\log n) \subset O(n) \subset O(n^2) \subset O((3/2)^n).$$

Solución al Problema 1.15 (☹)

Para resolver la ecuación

$$T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1} T(i)\right) + cn\ ,$$

siendo $T(0) = 0$, podemos reescribirla como:

$$nT(n) = \sum_{i=0}^{n-1} T(i) + cn^2 \qquad [1.7]$$

Por otro lado, para $n–1$ obtenemos:

$$(n-1)T(n-1) = \sum_{i=0}^{n-2} T(i) + c(n-1)^2 \qquad [1.8]$$

Restando [1.7] y [1.8]:

$$nT(n) - nT(n-1) + T(n-1) = T(n-1) + c(2n-1) \Rightarrow nT(n) = nT(n-1) + c(2n-1) \Rightarrow$$

$$T(n) = T(n-1) + c(2-1/n).$$

Desarrollando telescópicamente la ecuación en recurrencia:

$$\begin{aligned} T(n) &= T(n-1) + c(2 - 1/n) = \\ &= T(n-2) + c(2 - 1/(n-1)) + c(2 - 1/n) = \\ &= T(n-3) + c(2 - 1/(n-2)) + c(2 - 1/(n-1)) + c(2 - 1/n) = \\ &\dots\dots\dots\dots \\ &= T(0) + c\sum_{i=1}^{n}\left(2 - \frac{1}{i}\right) = \\ &= c\sum_{i=1}^{n}\left(2 - \frac{1}{i}\right) \end{aligned}$$

ya que teníamos que $T(0) = 0$. Veamos cual es el orden de $T(n)$:

a) Como $(2–1/i) \le 2$ para todo $i > 0$, $T(n) \le c\sum_{i=1}^{n} 2 = 2cn \Rightarrow T(n) \in O(n)$.

b) Como $(2–1/i) \ge 1$ para todo $i > 0$, $T(n) \ge c\sum_{i=1}^{n} 1 = cn \Rightarrow T(n) \in \Omega(n)$.

Por tanto, $T(n) \in \Theta(n)$.

Solución al Problema 1.16

Función *BuscBin* (😐)

a) Para determinar su tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

   – En la línea (1) se ejecutan la comparación del *IF* (1 OE), y un acceso a un vector (1 OE), una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
   – En la línea (3) se realizan 3 OE (suma, división y asignación).
   – En la línea (4) hay un acceso a un vector (1 OE) y una comparación (1 OE), y además 1 OE en caso de que la condición del *IF* sea verdadera.
   – En la línea (5) hay un acceso a un vector (1 OE) y una comparación (1 OE).
   – Las líneas (6) y (8) efectúan $3+T(n/2)$ cada una: una operación aritmética (incremento o decremento de 1), una llamada a la función *BuscBin* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con la mitad de los elementos y un *RETURN* (1 OE).

   Por tanto obtenemos la ecuación en recurrencia $T(n) = 11 + T(n/2)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k = T(2^k)$ obtenemos

$$t_k - t_{k-1} = 11,$$

   ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 k + c_2$$

   y, deshaciendo los cambios,

$$T(n) = c_1 \log n + c_2.$$

   Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(2)$, que podemos calcular apoyándonos en la expresión de la ecuación en recurrencia: $T(2) = 11 + 4 = 15$. Finalmente obtenemos

$$T(n) = 11\log n + 4 \in \Theta(\log n)$$

b) La recursión de este programa, por tratarse de un caso de recursión de cola, puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función. La condición de terminación del bucle puede ser tomada del caso base de la función recursiva y el cuerpo de dicho bucle consiste en una preparación de los argumentos de la función recursiva y el cálculo que ésta realiza:

```
PROCEDURE BuscBIt(a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
  VAR mitad:CARDINAL;
BEGIN
  WHILE (prim<ult) DO                      (* 1 *)
    mitad:=(prim+ult)DIV 2;                 (* 2 *)
    IF x=a[mitad] THEN RETURN TRUE          (* 3 *)
    ELSIF (x<a[mitad]) THEN                 (* 4 *)
      ult:=mitad-1                          (* 5 *)
    ELSE                                    (* 6 *)
```

```
          prim:=mitad+1                      (* 7 *)
        END                                  (* 8 *)
      END;                                   (* 9 *)
      RETURN x=a[ult]                        (* 10 *)
    END BuscBIt;
```

c) Para el cálculo del tiempo de ejecución y la complejidad de la función no recursiva podemos seguir un proceso análogo al que seguimos para la función *Algoritmo2* (en el problema 1.5). Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan en cada una de las líneas:

– En la línea (1) se efectúa la condición del bucle, que supone 1 OE (la comparación).
– Las líneas (2) a (9) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2, 2, 0, 2, 0 y 0 OE respectivamente.
– Por último, la línea (10) supone 3 OE. A ella se llega cuando la condición del bucle deja de verificarse.

El bucle se repite hasta que su condición sea falsa, acabando la función al ejecutarse la línea (10). Cada iteración del bucle está compuesta por las líneas (1) a (9), junto con una ejecución adicional de la línea (1) que es la que ocasiona la salida del bucle. En cada iteración se reduce a la mitad los elementos a considerar, por lo que el bucle se repite log$n$ veces. Por tanto, en el peor caso,

$$T(n) = \left(\left(\sum_{i=1}^{\log n}(1+3+2+2+2)\right)+1\right)+3 = 10\log n + 4 \in \Theta(\log n).$$

Como puede verse, el tiempo de ejecución de ambas funciones es prácticamente igual, lo que a priori implica que cualquiera de las dos pueden usarse indistintamente. Sin embargo, hay que tener en cuenta la mayor complejidad espacial que siempre suponen los procedimientos recursivos por la utilización de la pila, lo que hace que ante una igualdad de tiempos de ejecución, los procedimientos iterativos sean preferibles frente a los recursivos. Pero no sólo la complejidad ha de ser tenida en cuenta para la elección del algoritmo. La claridad y sencillez del código es un factor también a considerar, pues ello va a implicar una mejor legibilidad y una depuración del programa y mantenimiento más fácil, aspectos todos ellos muy importantes.

Función *Sumadigitos* (😐)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
– En la línea (2) se efectúa una división (1 OE), una llamada a la función *Sumadigitos* (1 OE), más lo que tarda ésta con un décimo del tamaño de su entrada, una suma (1 OE), un resto (1 OE), y un *RETURN* (1 OE).

Llamando n al parámetro *num* de la función, obtenemos la ecuación en recurrencia $T(n) = 6 + T(n/10)$, con la condición inicial $T(1) = 2$.

Para resolverla hacemos los cambios $n = 10^k$ (o, lo que es igual, $k = \log_{10}n$) y $t_k = T(10^k)$ y obtenemos

$$t_k - t_{k-1} = 6,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2k.$$

Deshaciendo los cambios,

$$T(n) = c_1 + c_2\log_{10}n.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(1) = 2$, junto con el valor de $T(10)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia: $T(10) = 6 + 2 = 8$. Finalmente obtenemos

$$T(n) = 6 \log_{10}n + 2 \in \Theta(\log n)$$

Como vemos, en esta caso la complejidad de la función depende del logaritmo en base 10 de su parámetro *num* (esto es, de su número de dígitos).

b) La recursión de este algoritmo puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función, cuya condición de terminación puede ser tomada del caso base de la función recursiva, y cuyo cuerpo consiste en los cálculos que ésta realiza, junto con una preparación de los argumentos de la siguiente llamada. En concreto, el algoritmo que implementa el algoritmo no recursivo es el siguiente:

```
PROCEDURE Sumadigitos_it(num:CARDINAL):CARDINAL;
 VAR s:CARDINAL;
BEGIN
 s:=num MOD 10;                          (* 1 *)
 WHILE num>=10 DO                        (* 2 *)
    num:=num DIV 10;                     (* 3 *)
    s:=s+(num MOD 10)                    (* 4 *)
 END;                                    (* 5 *)
 RETURN s                                (* 6 *)
END Sumadigitos_it;
```

c) Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecutan 2 OE (un resto y una asignación).
– En la línea (2) se efectúa la condición del bucle, que supone 1 OE.
– Las líneas (3) y (4) componen el cuerpo del bucle, y contabilizan 2 y 3 OE respectivamente.
– Por último, la línea (6) supone 1 OE. A ella se llega cuando la condición del bucle deja de verificarse.

Con esto, se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (6). Cada iteración del bucle está compuesta por las líneas (2) a (4), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se diezman los elementos a considerar, por lo que el bucle se repite $\log_{10}n$ veces.

Por tanto, $T(n) = 2 + \left(\left(\sum_{i=1}^{\log_{10} n}(1+2+3)\right)+1\right)+1 = 6\log_{10} n + 4 \in \Theta(\log n)$.

Llegados a este punto vemos que ocurre lo mismo que en el algoritmo anterior. Los tiempos de ejecución de las funciones recursiva e iterativa son similares. Es por tanto una decisión del usuario decantarse por el diseño generalmente más robusto ofrecido por los algoritmos recursivos, frente a la menor complejidad espacial que presentan los iterativos al no utilizar la pila de recursión.

Solución al Problema 1.17 (☺)

a) Para determinar el tiempo de ejecución del algoritmo, calcularemos primero el número de operaciones elementales (OE) que se realizan:
– En la línea (1) se puede ejecutar o sólo la condición (si ésta es falsa) con un total de 1 OE, o bien la sentencia entera, con un total de 3 OE.
– Las líneas (2) y (3) están compuestas por operaciones aritméticas y asignaciones, y se realizan un total de 3 OE en cada una.
– La línea (4) tiene tres partes: un acceso a *a*[*mitad*], que supone 1 OE; la llamada a *Raro(a,prim,prim+terc)*, que supone 2 + $T(n/3)$ (1 de la suma de *prim* y *terc*, 1 OE de la llamada a *Raro*, y luego el tiempo de ejecución de *Raro* para un tercio del número de elementos con los que fue invocada la función original); y la llamada a *Raro(a,ult–terc,ult)*, que supone 2 + $T(n/3)$ OE por la misma razón. El resultado de las tres partes ha de sumarse (lo que supone 2 OE) y luego hacer un *RETURN* (1 OE). En resumen, en la línea (4) se ejecutan un total de $1 + 2 + T(n/3) + 2 + T(n/3) + 2 + 1 = 8 + 2T(n/3)$ OE.

Con esto:
1. Si $n = 1$ (caso base), se ejecuta sólo la línea (1) y por tanto $T(1) = 3$.
2. Si $n = 3^k$, con $k > 0$, se ejecuta sólo la condición del *IF* (1) y luego el resto de las líneas (2) a (4), con lo cual $T(n) = 15 + 2T(n/3)$.

Tenemos por tanto el tiempo de ejecución del algoritmo definido mediante una ecuación en recurrencia. Para resolverla, haciendo el cambio $n = 3^k$ queda

$$T(3^k) = 2T(3^{k-1}) + 15$$

y llamando t_k a $T(3^k)$ obtenemos

$$t_k = t_{k-1} + 15,$$

ecuación en recurrencia no homogénea de ecuación característica $(x-2)(x-1) = 0$ y consecuentemente

$$t_k = c_1 2^k + c_2 1^k = c_1 2^k + c_2.$$

Cambiando entonces t_k por $T(3^k)$ queda

$$T(3^k) = c_1 2^k + c_2,$$

y deshaciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$), obtenemos finalmente

$$T(n) = c_1 2^{\log_3 n} + c_2 = c_1 n^{\log_3 2} + c_2.$$

Para calcular las constantes necesitamos resolver un sistema de dos ecuaciones con dos incógnitas (c_1 y c_2) basándonos en dos condiciones iniciales de la ecuación en recurrencia. Como de partida sólo disponemos de una ($T(1) = 3$), necesitamos obtener otra. Para ello, apoyándonos en la definición recursiva de T y para $n = 3$, obtenemos

$$T(3) = 15 + 2T(n/3) = 15 + 2T(1) = 21.$$

Por tanto

$$\left.\begin{array}{r} 3 = T(1) = c_1 + c_2 \\ 21 = T(3) = 2c_1 + c_2 \end{array}\right\} \Rightarrow \left.\begin{array}{l} c_1 = 18 \\ c_2 = -15 \end{array}\right\}$$

Sustituyendo estos valores en la ecuación, obtenemos finalmente

$$T(n) = 18n^{\log_3 2} - 15.$$

b) $T(n) \in \Theta(n^{\log_3 2})$.

Para justificarlo, basándonos en las propiedades de Θ, basta ver que

$$\lim_{n\to\infty} \frac{T(n)}{n^{\log_3 2}}$$

existe, es acotado y distinto de cero. Pero eso es cierto ya que

$$\lim_{n\to\infty}\frac{T(n)}{n^{\log_3 2}}=\lim_{n\to\infty}\frac{18n^{\log_3 2}-15}{n^{\log_3 2}}=18.$$